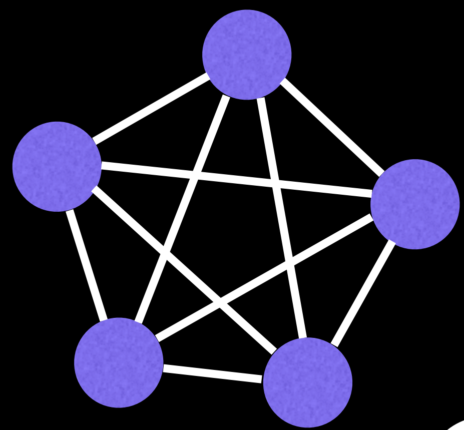
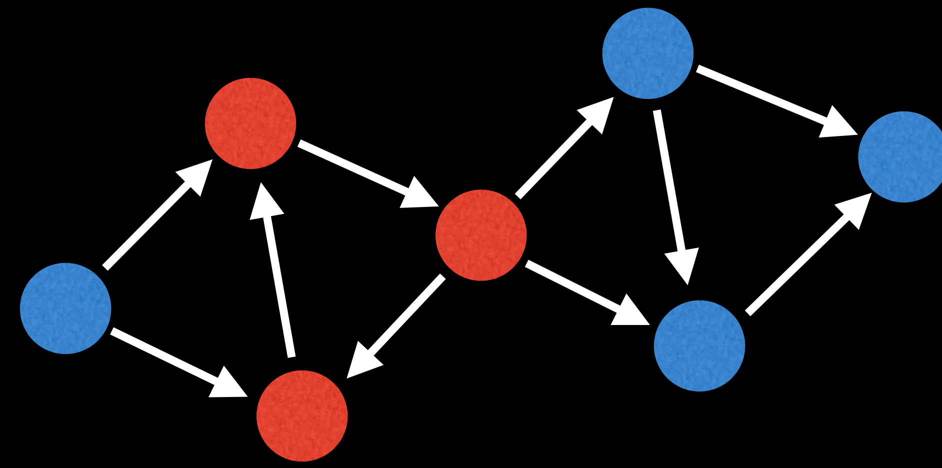
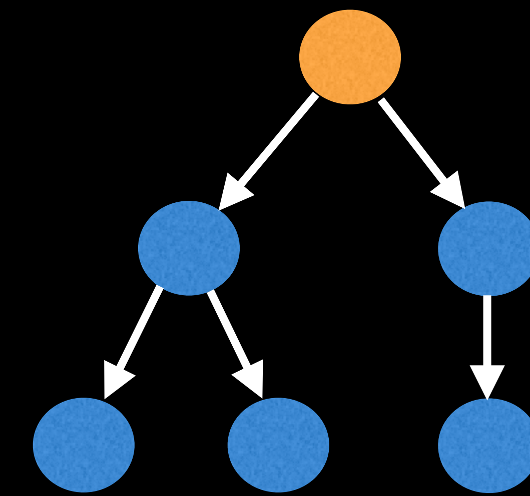
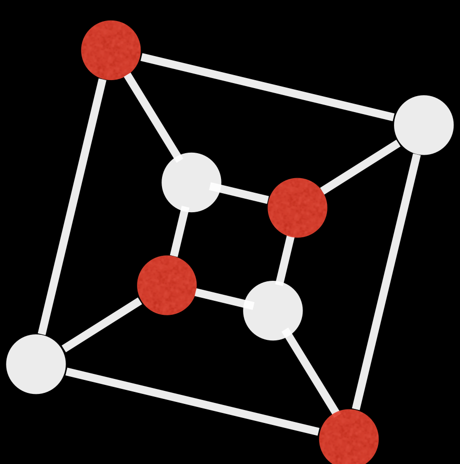
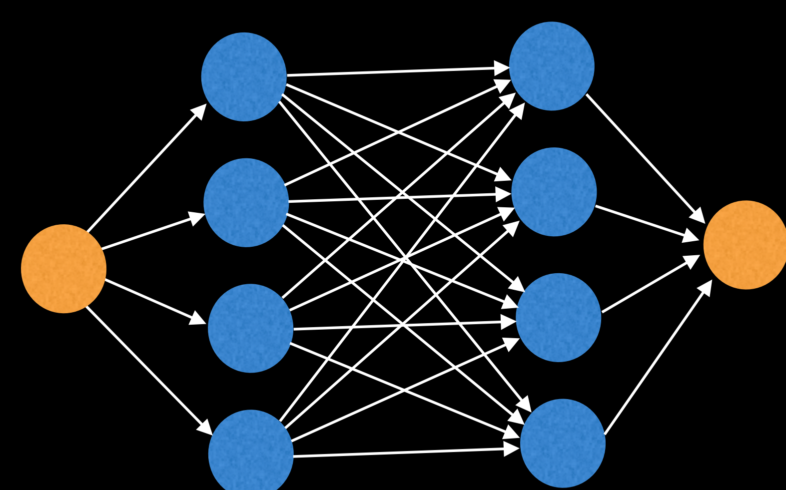


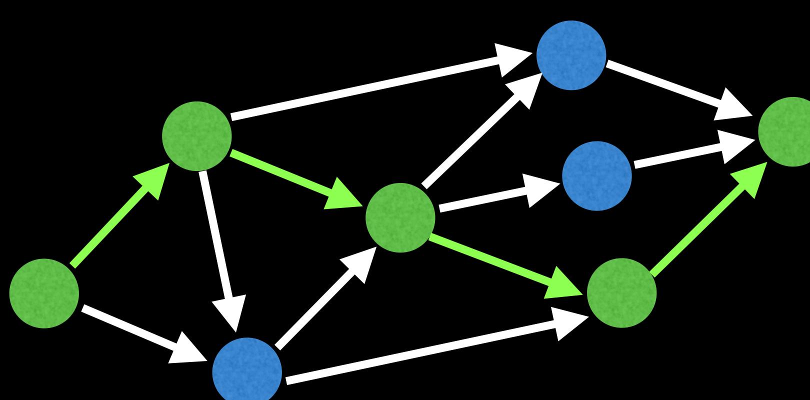




Graph Theory Video Series







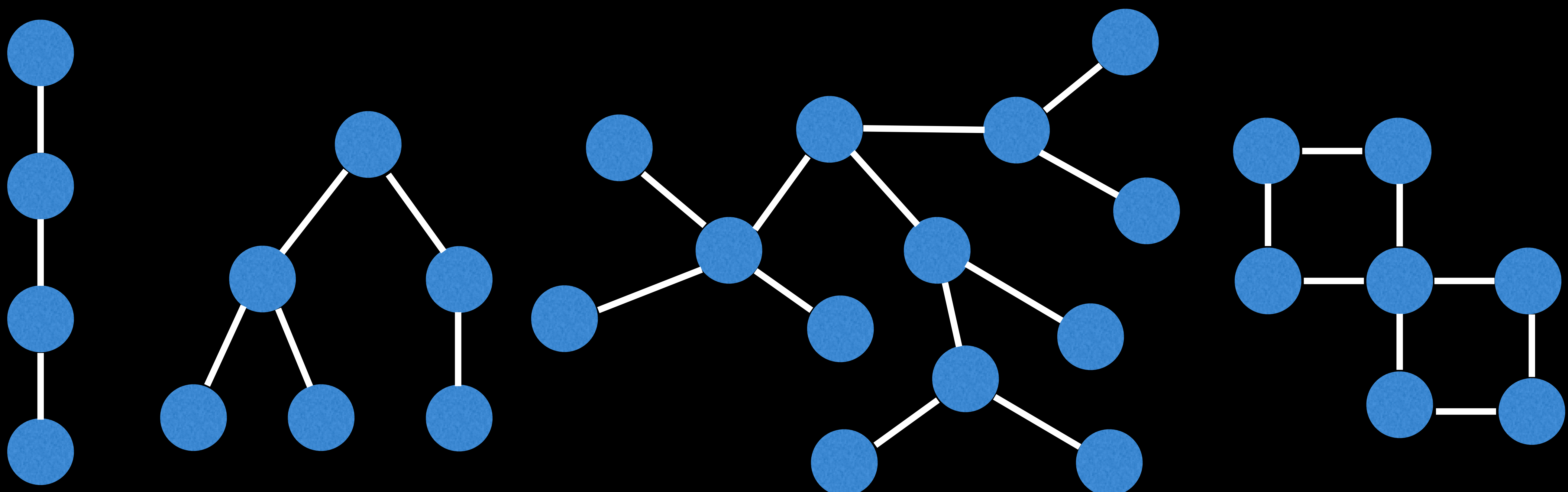
Storage and representation of trees

Definitions and storage representation

William Fiset

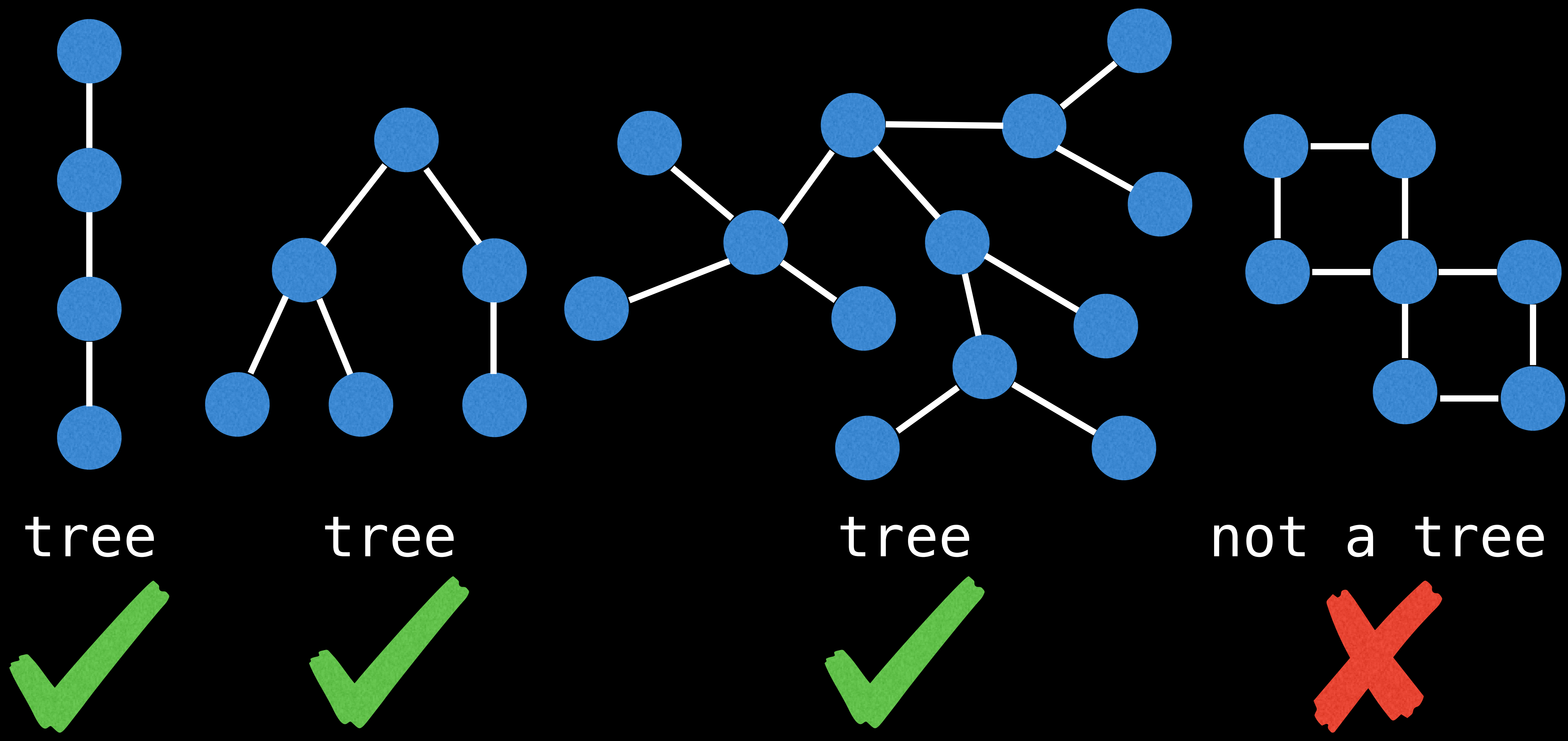
Trees!

What ***is*** a tree?



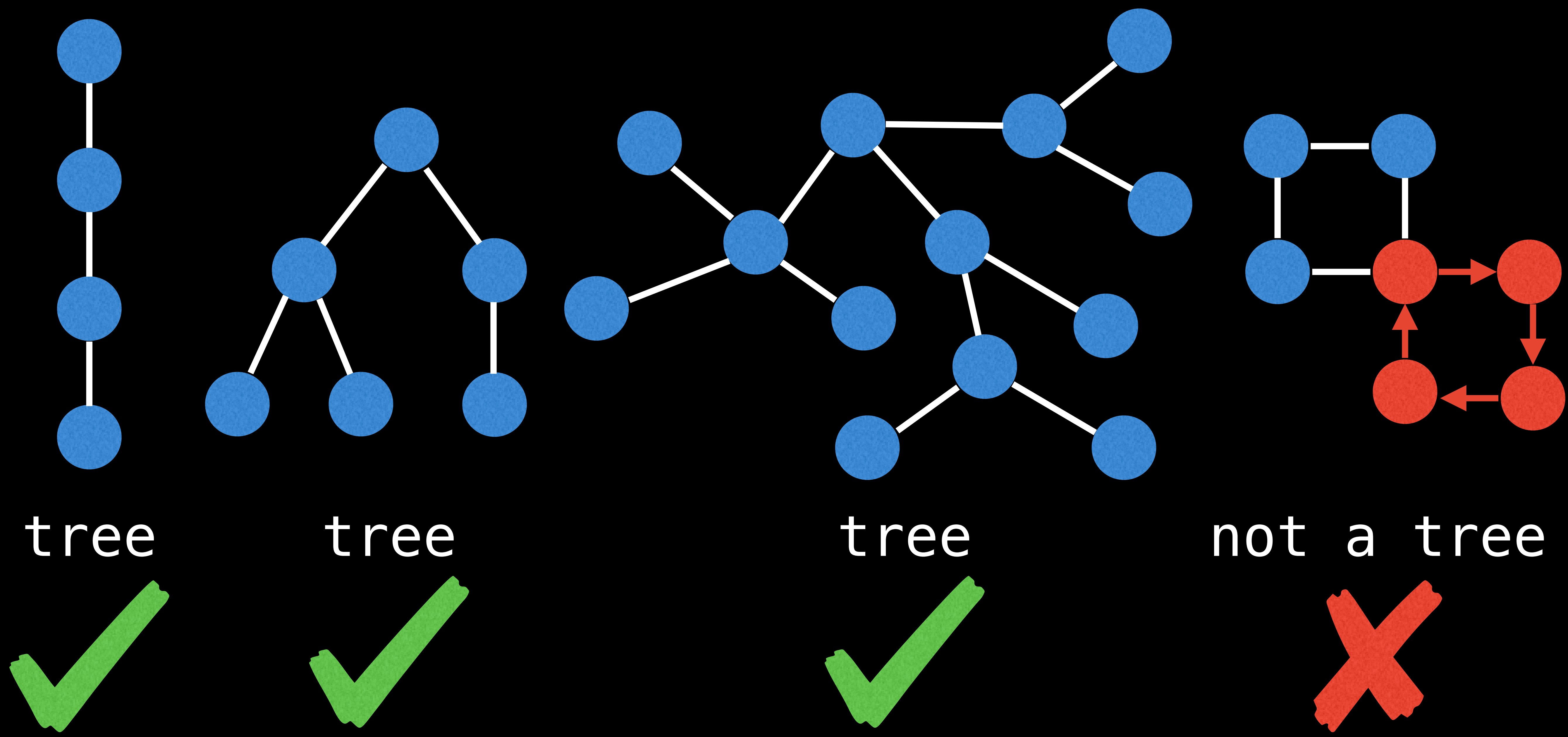
Trees!

What ***is*** a tree?



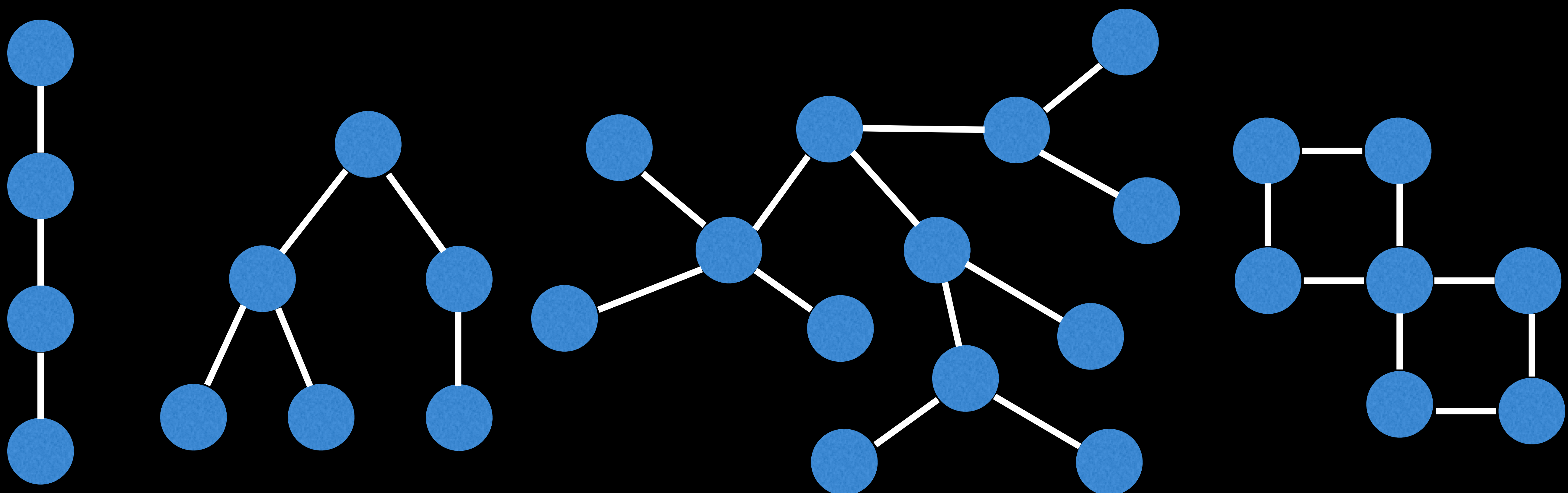
Trees!

A **tree** is a connected, undirected graph with **no cycles**.



Trees!

Equivalently, a **tree** it is a connected graph with N nodes and N–1 edges.



4 nodes
3 edges

6 nodes
5 edges

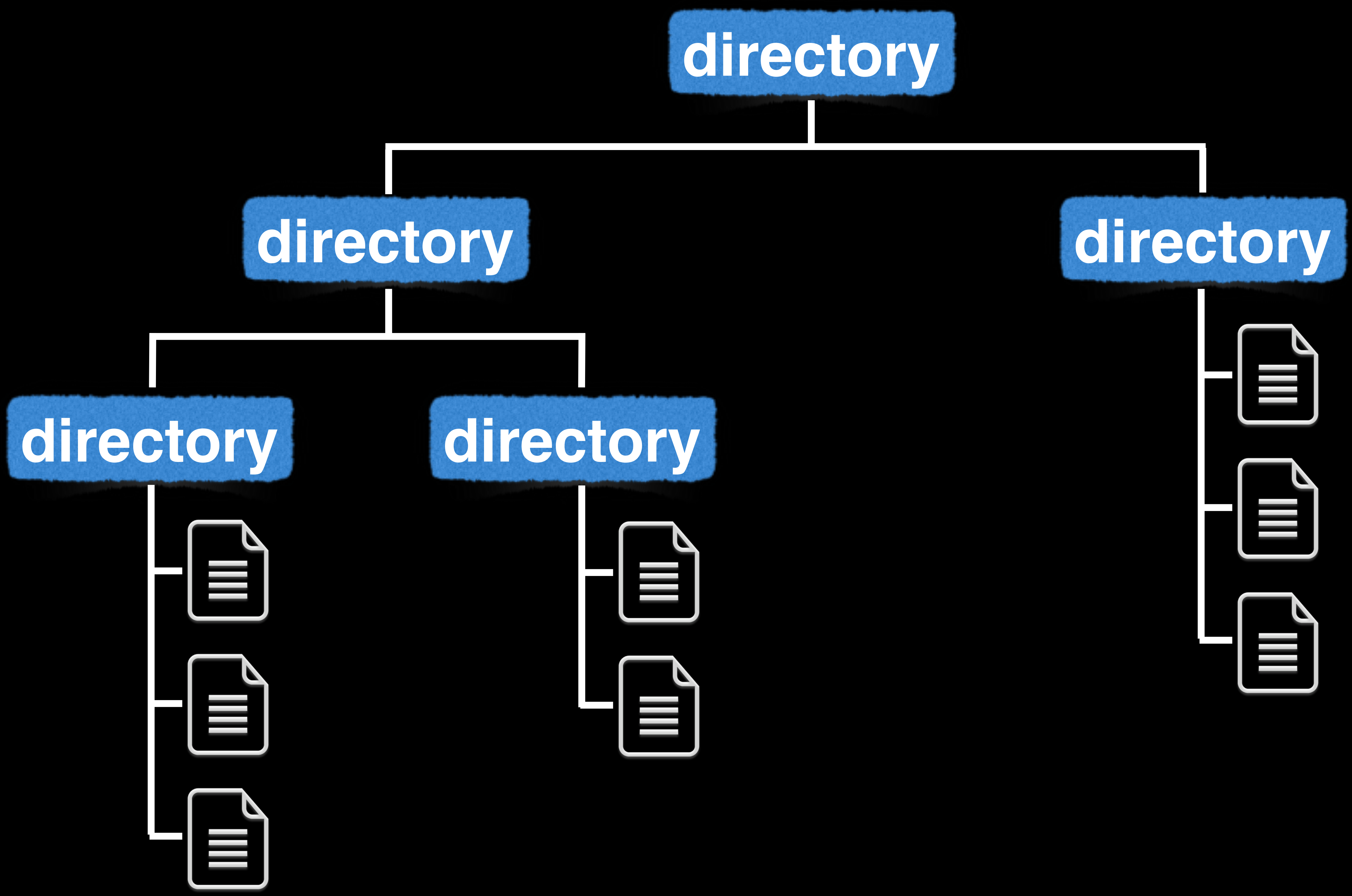
13 nodes
12 edges

7 nodes
8 edges

Trees out in the wild

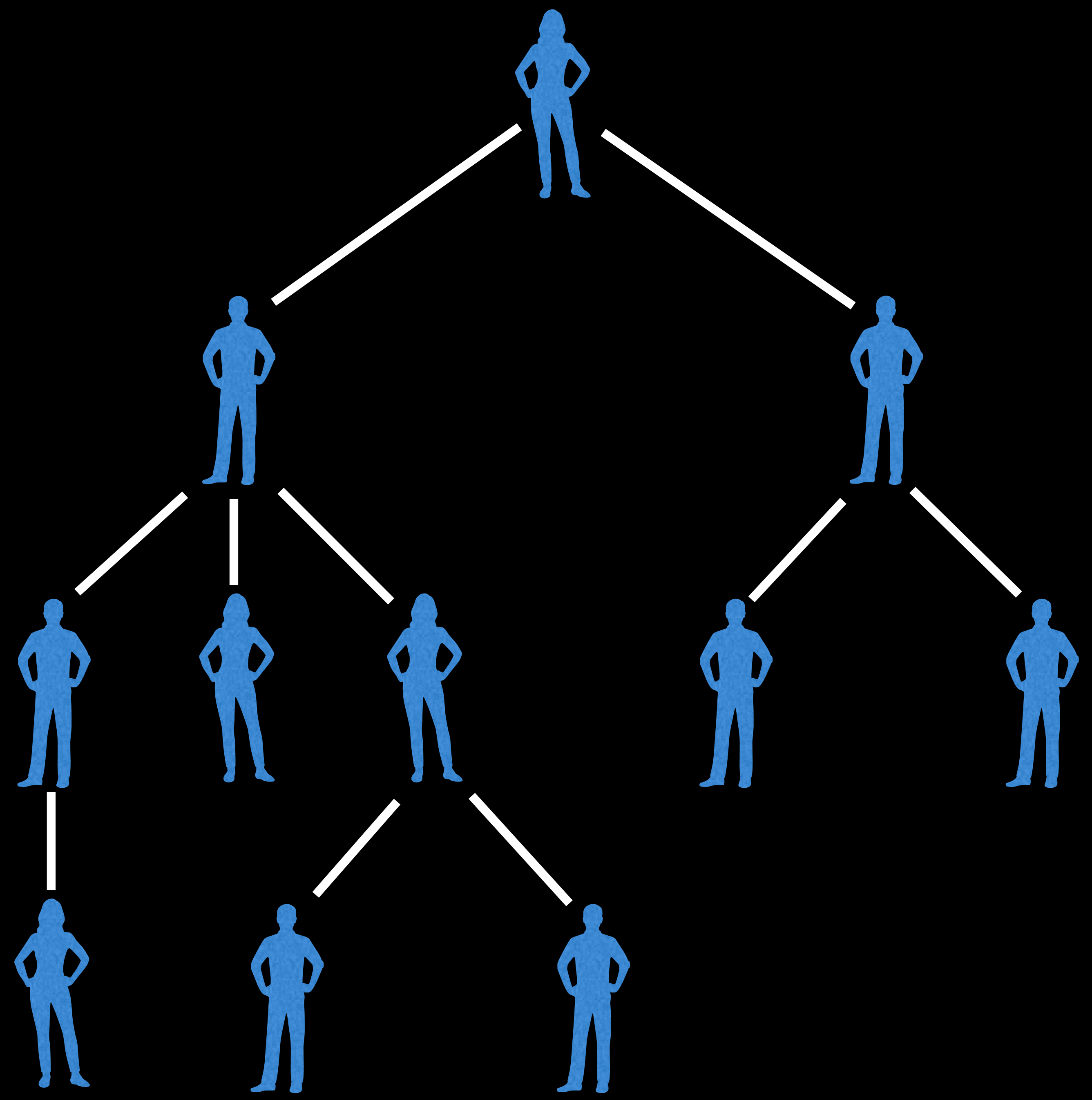
Trees out in the wild

Filesystem structures are inherently trees



Trees out in the wild

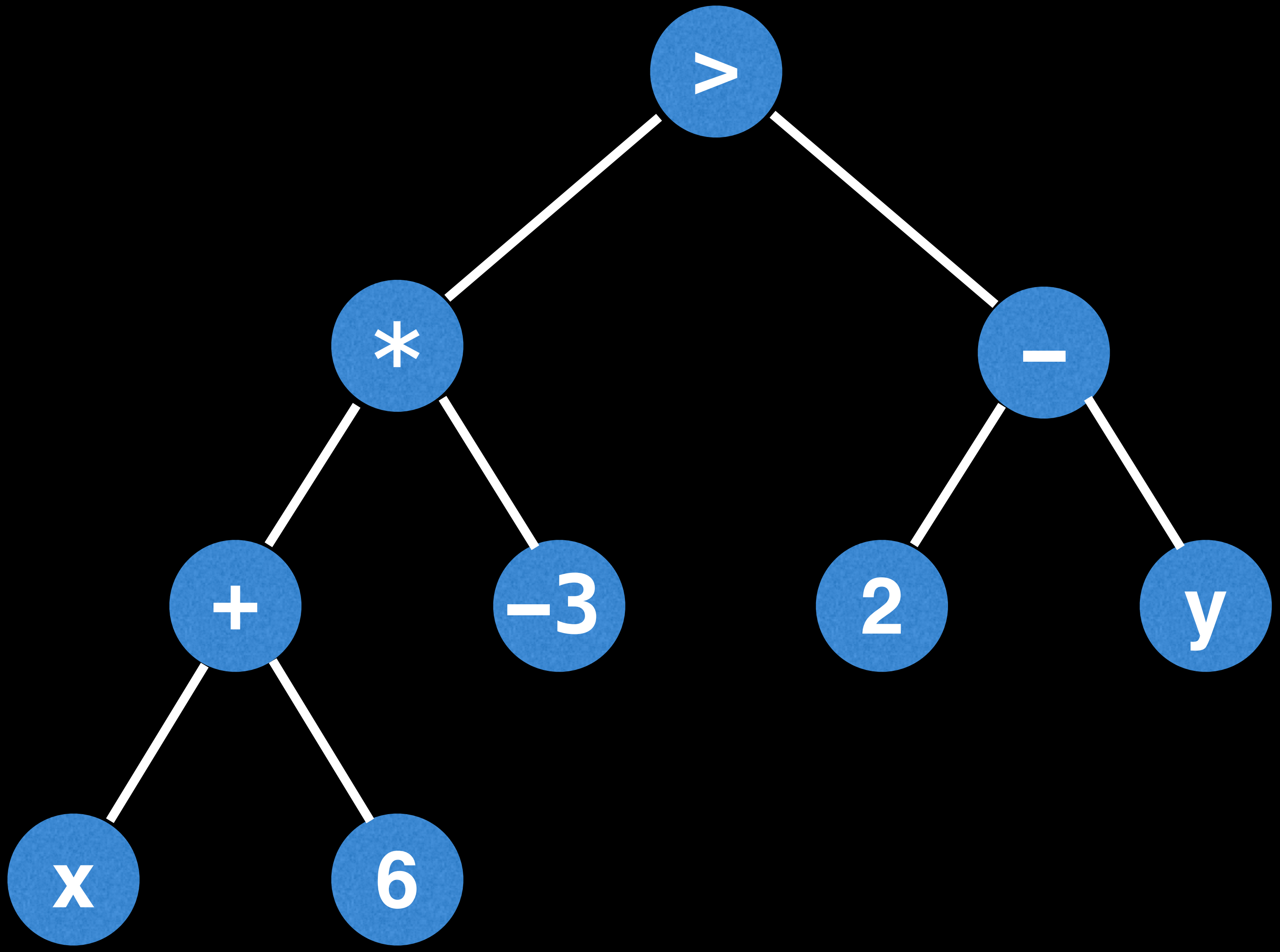
Social hierarchies



Trees out in the wild

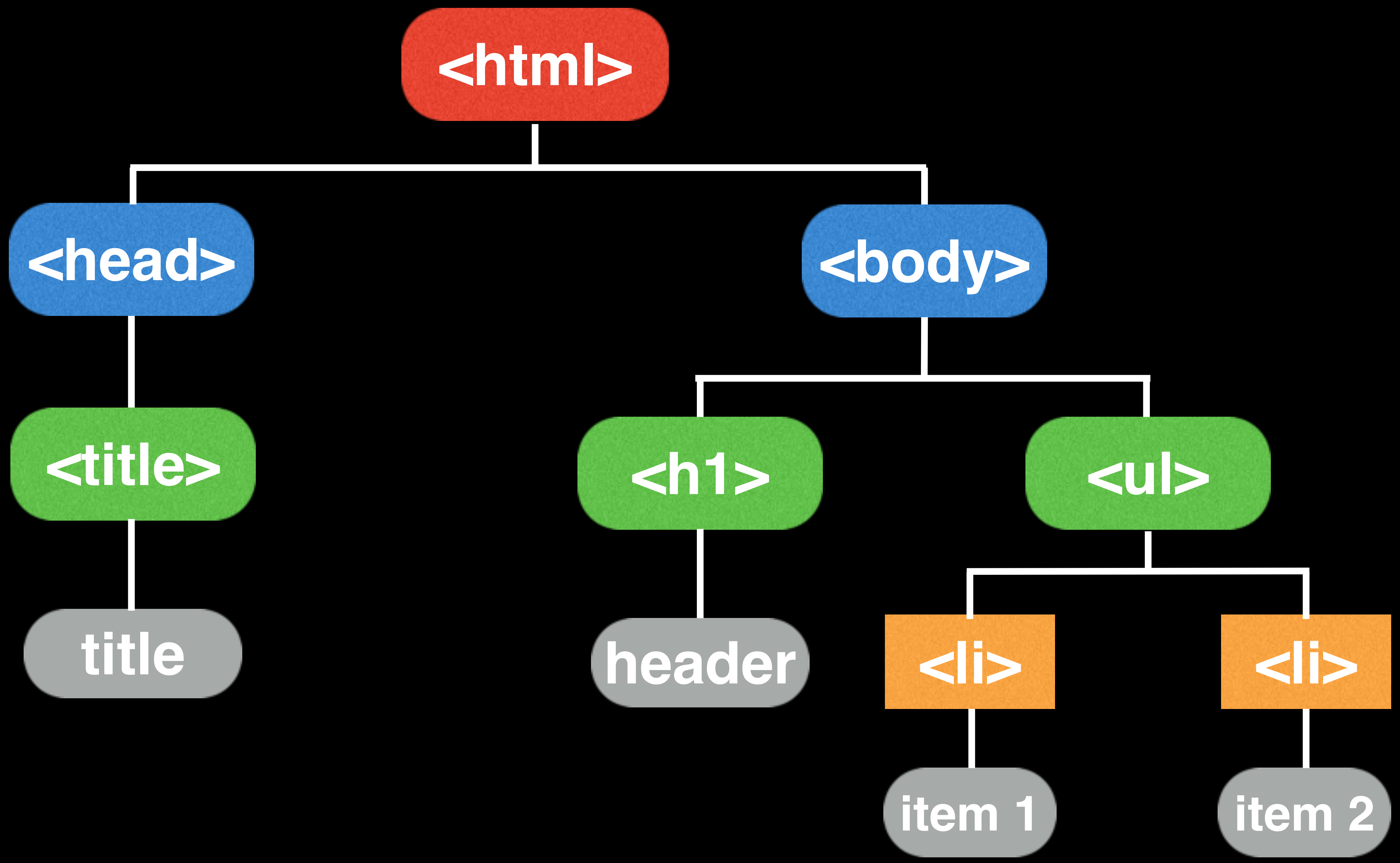
Abstract syntax trees to decompose source code and mathematical expressions for easy evaluation.

$$((x + 6) * -3) > (2 - y)$$



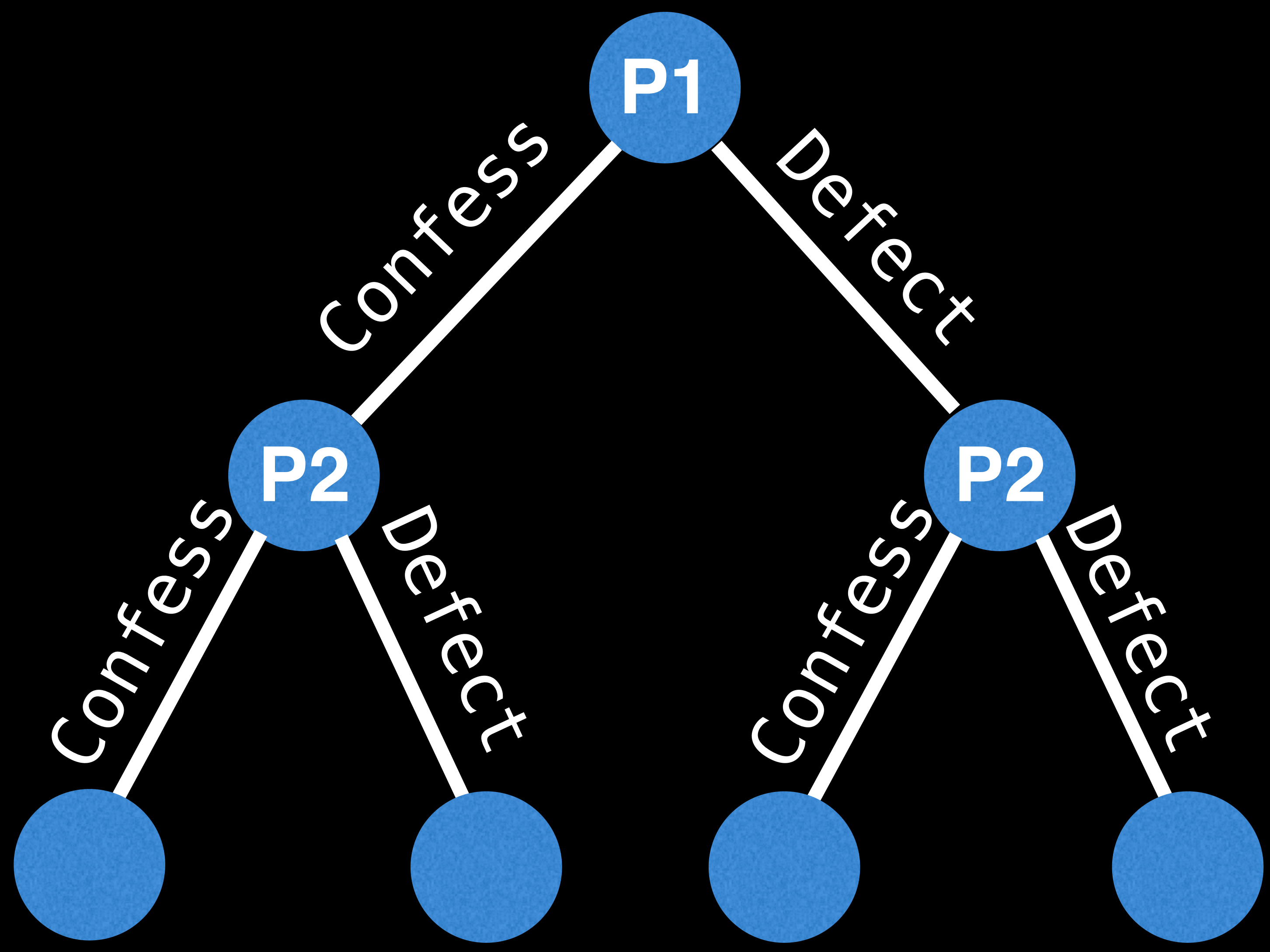
Trees out in the wild

Every webpage is a tree as an HTML DOM structure



Trees out in the wild

The decision outcomes in game theory are often modeled as trees for ease of representation.



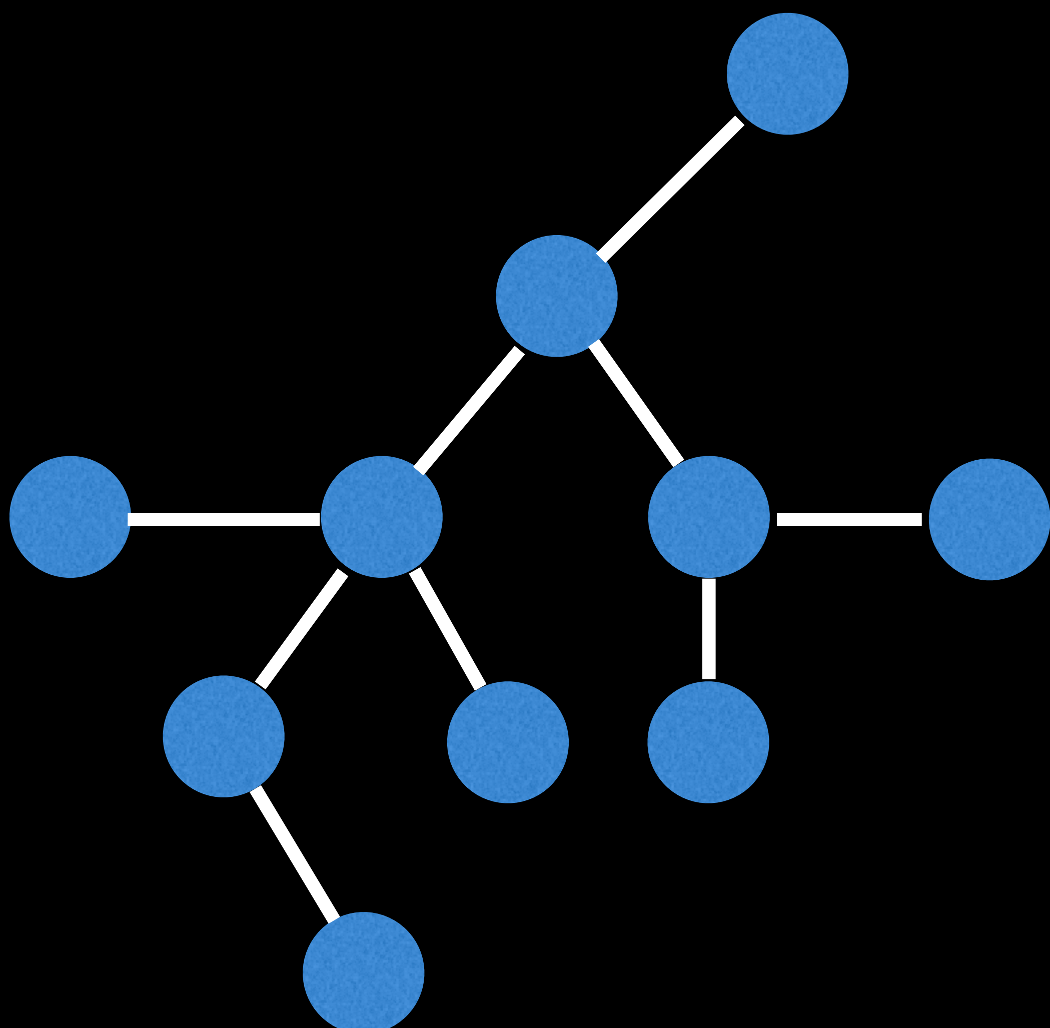
Tree of the prisoner’s dilemma

Trees out in the wild

There are many many more applications…

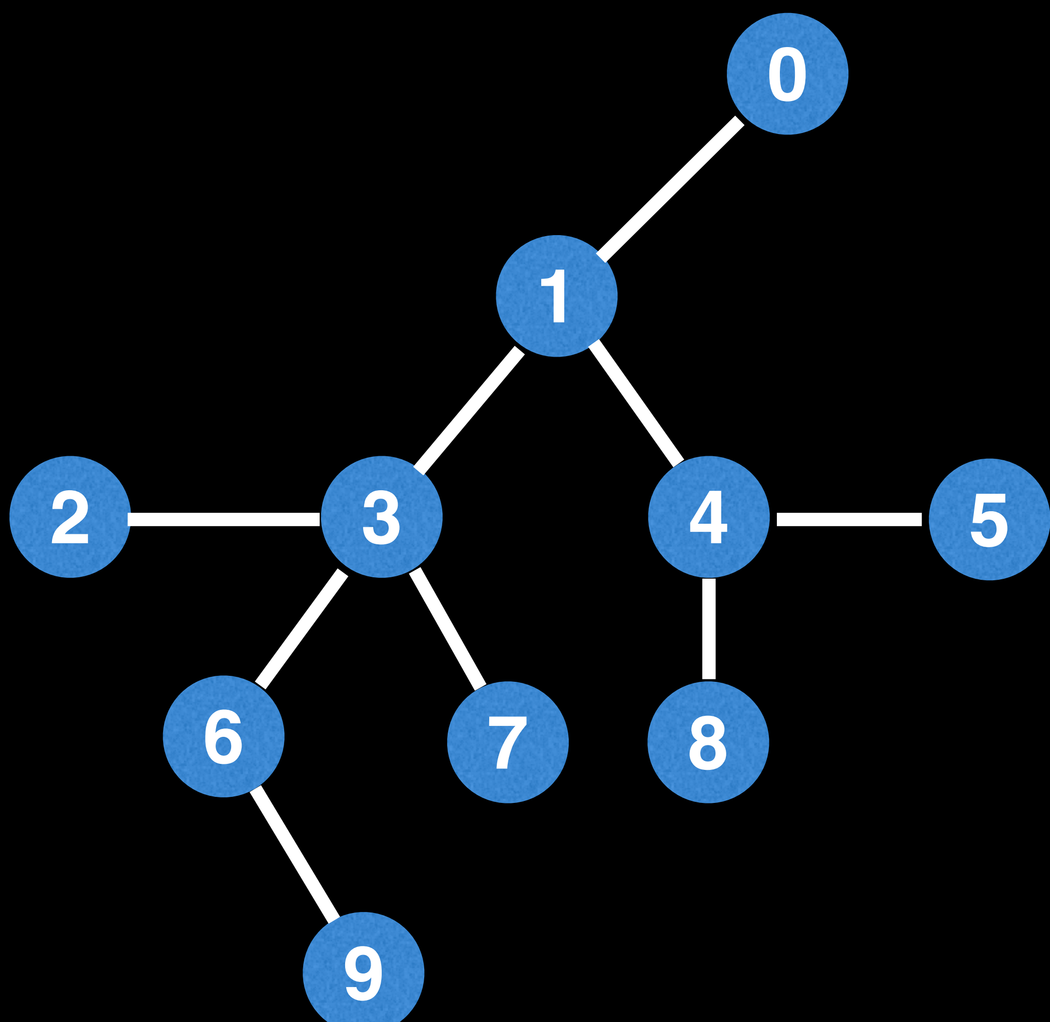
- Family trees
- File parsing/HTML/JSON/Syntax trees
- Many data structures use/are trees:
  - AVL trees, B-tree, red-black trees, segment trees, fenwick trees, treaps, suffix trees, tree maps/sets, etc…
- Game theory decision trees
- Organizational structures
- Probabilty trees
- Taxonomies
- etc…

Storing undirected trees

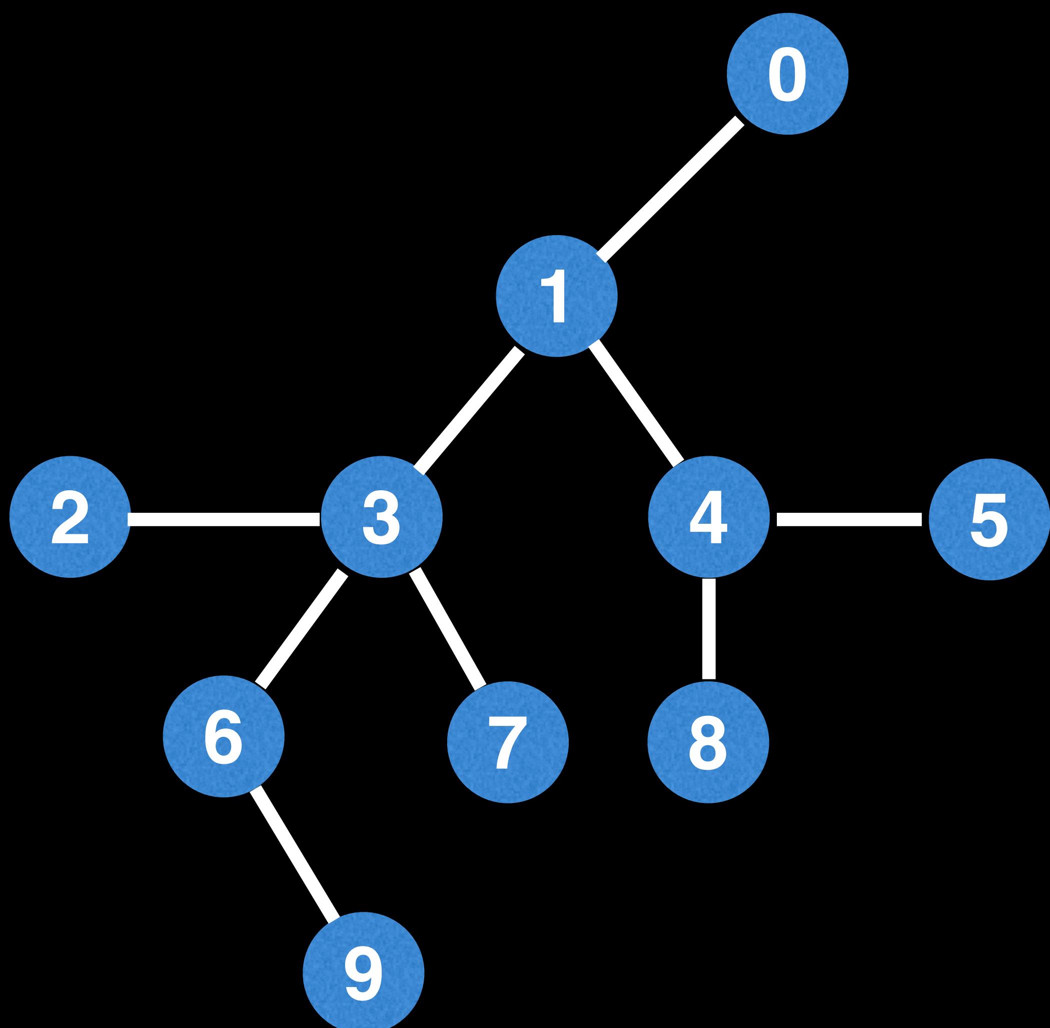


Storing undirected trees

Start by labelling the tree
nodes from [0, n)



Storing undirected trees

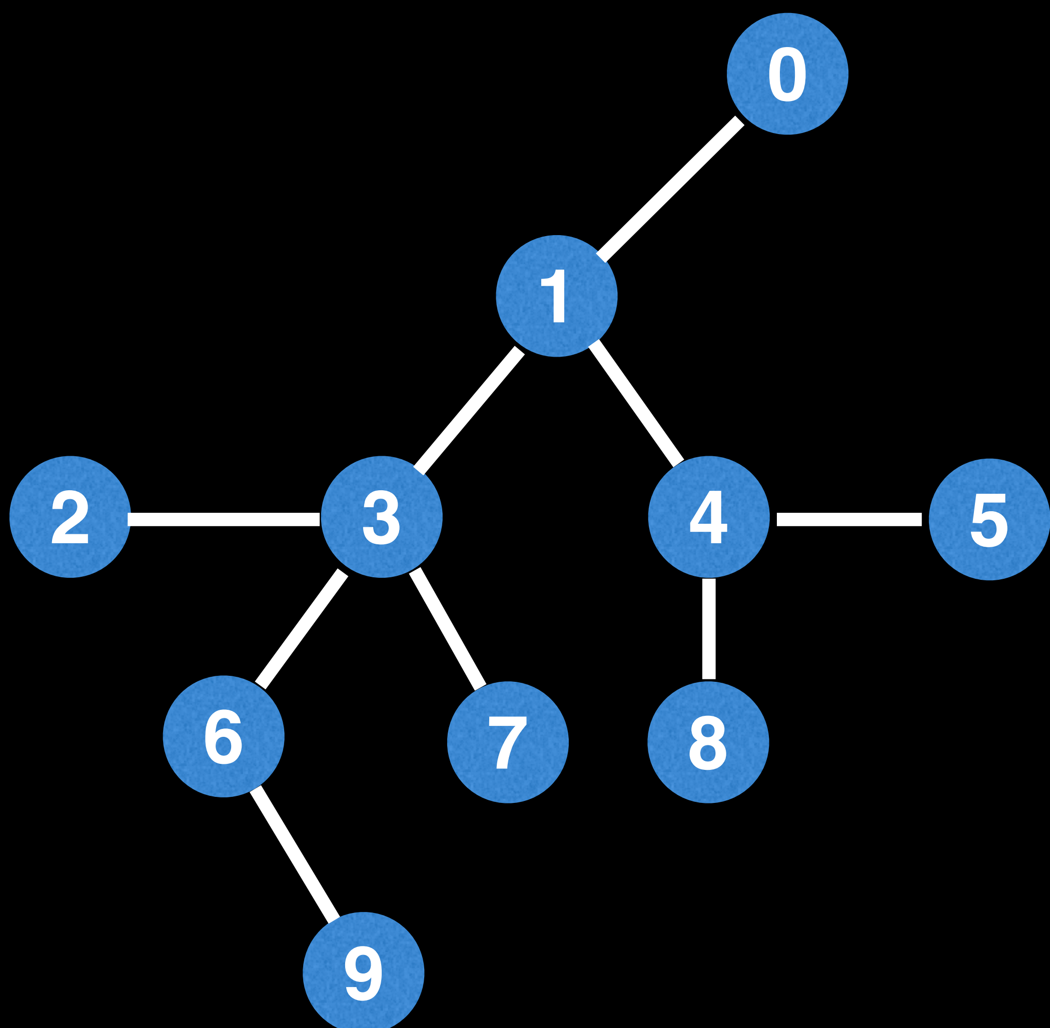


edge list storage representation:

```
[(0, 1),
 (1, 4),
 (4, 5),
 (4, 8),
 (1, 3),
 (3, 7),
 (3, 6),
 (2, 3),
 (6, 9)]
```

pro: simple and easy to iterate over.

Storing undirected trees

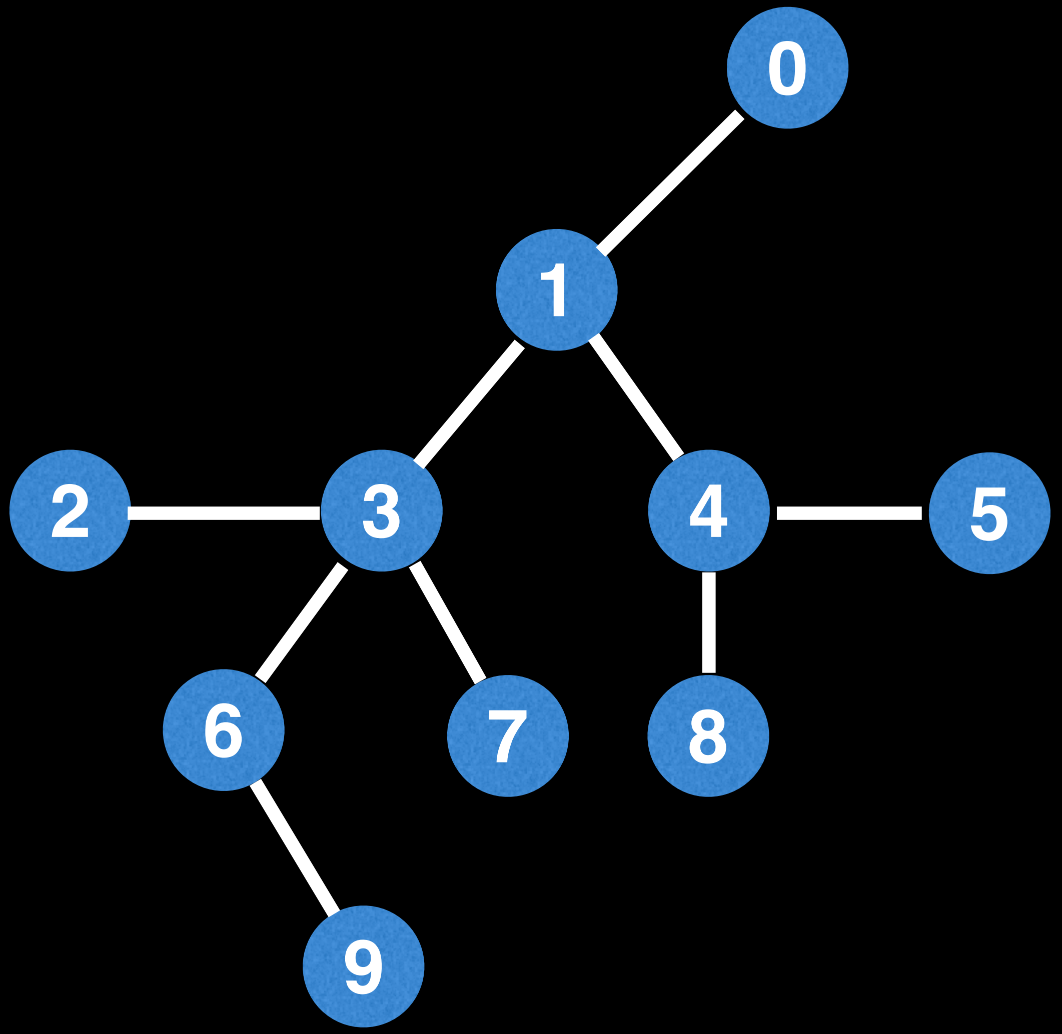


edge list storage representation:

```
[(0, 1),
 (1, 4),
 (4, 5),
 (4, 8),
 (1, 3),
 (3, 7),
 (3, 6),
 (2, 3),
 (6, 9)]
```

con: storing a tree as a list lacks the structure to efficiently query all the neighbors of a node.

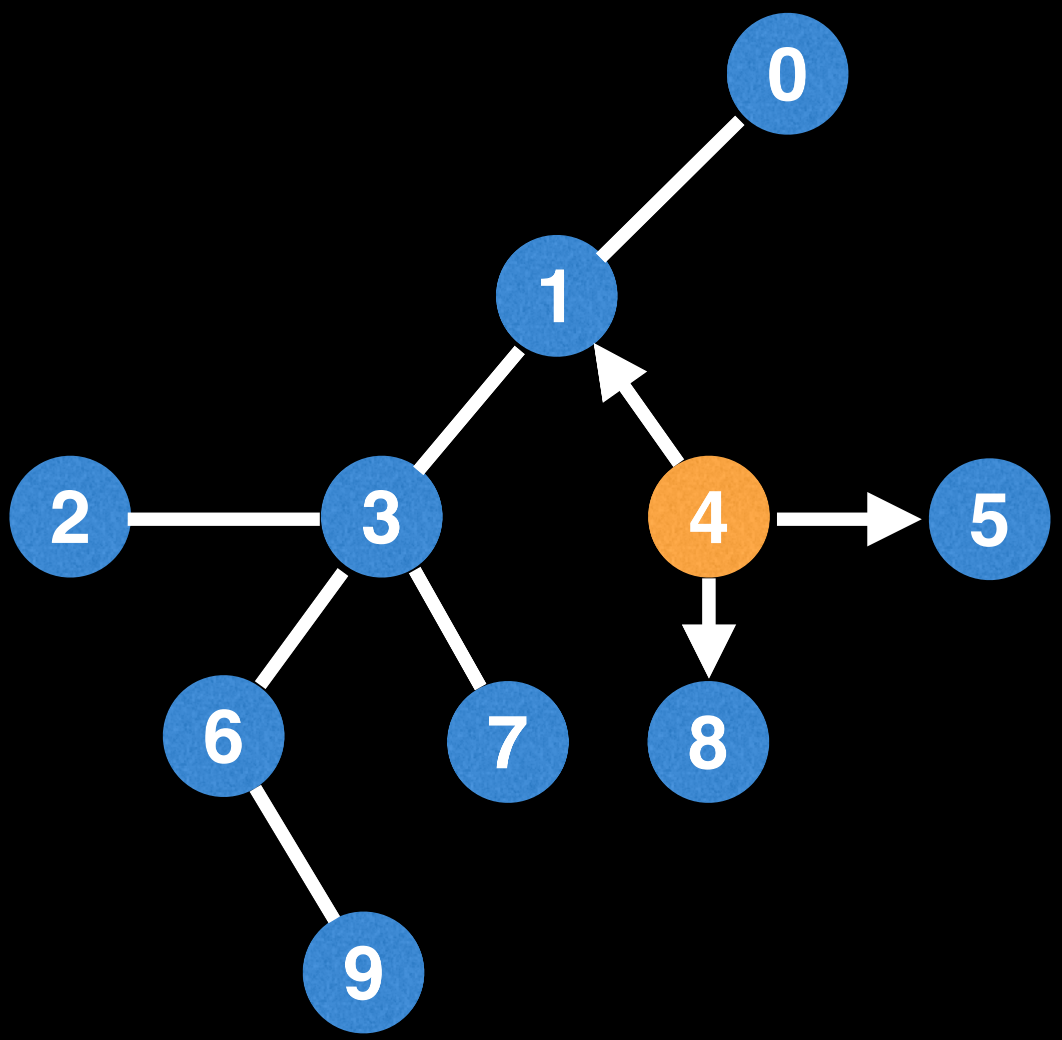
Storing undirected trees



adjacency list representation

```
0 -> [1]
1 -> [0,3,4]
2 -> [3]
3 -> [1,2,6,7]
4 -> [1,5,8]
5 -> [4]
6 -> [3,9]
7 -> [3]
8 -> [4]
9 -> [6]
```

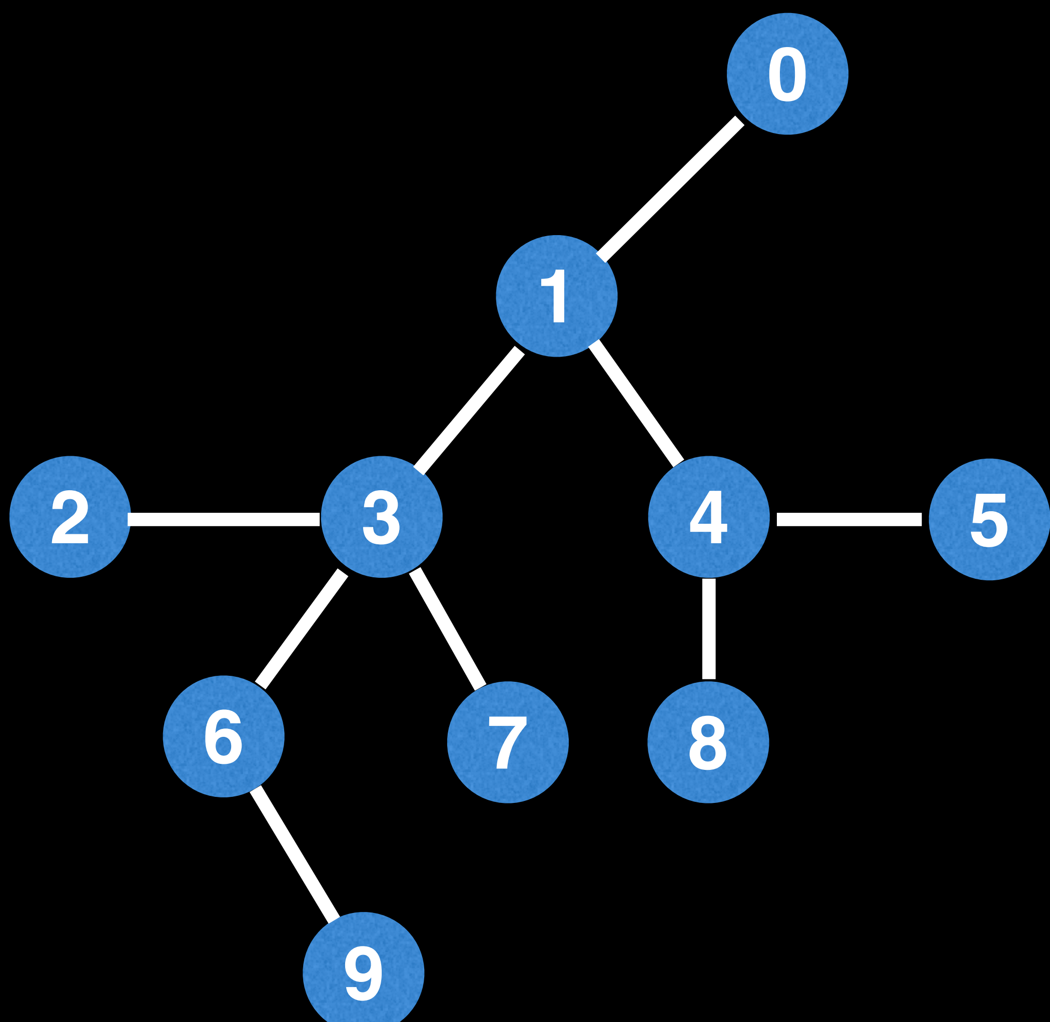
Storing undirected trees



adjacency list representation

```
0 -> [1]
1 -> [0,3,4]
2 -> [3]
3 -> [1,2,6,7]
4 -> [1,5,8]
5 -> [4]
6 -> [3,9]
7 -> [3]
8 -> [4]
9 -> [6]
```

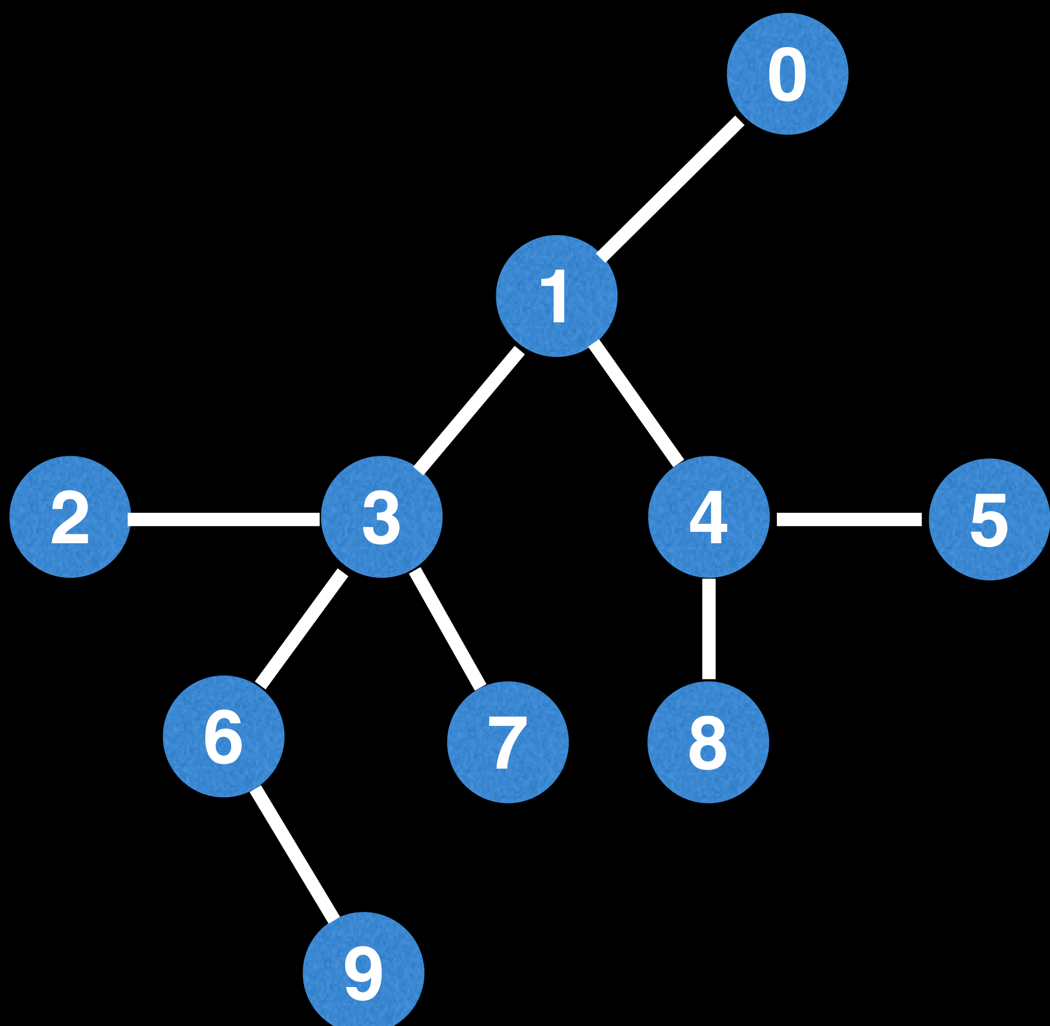
Storing undirected trees



adjacency matrix representation

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	0	0	0	0
1	1	0	0	1	1	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0
3	0	1	1	0	0	0	1	1	0	0
4	0	1	0	0	0	1	0	0	1	0
5	0	0	0	0	1	0	0	0	0	0
6	0	0	0	1	0	0	0	0	0	1
7	0	0	0	1	0	0	0	0	0	0
8	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	1	0	0	0

Storing undirected trees



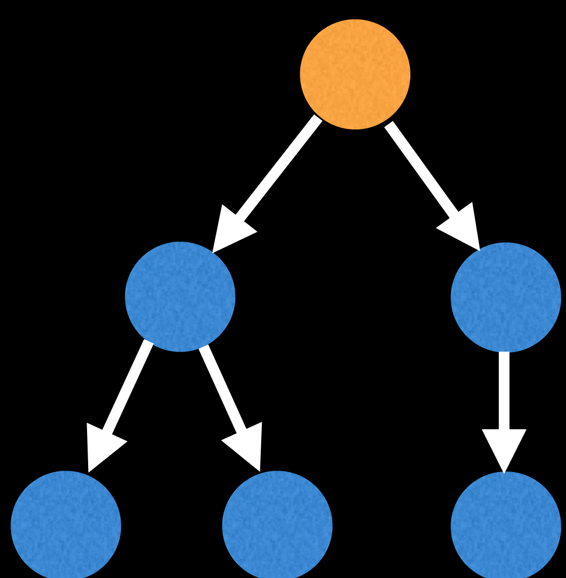
adjacency matrix representation

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	0	0	0	0
1	1	0	0	1	1	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0
3	0	1	1	0	0	0	1	1	0	0
4	0	1	0	0	0	1	0	0	1	0
5	0	0	0	0	1	0	0	0	0	0
6	0	0	0	1	0	0	0	0	0	1
7	0	0	0	1	0	0	0	0	0	0
8	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	1	0	0	0

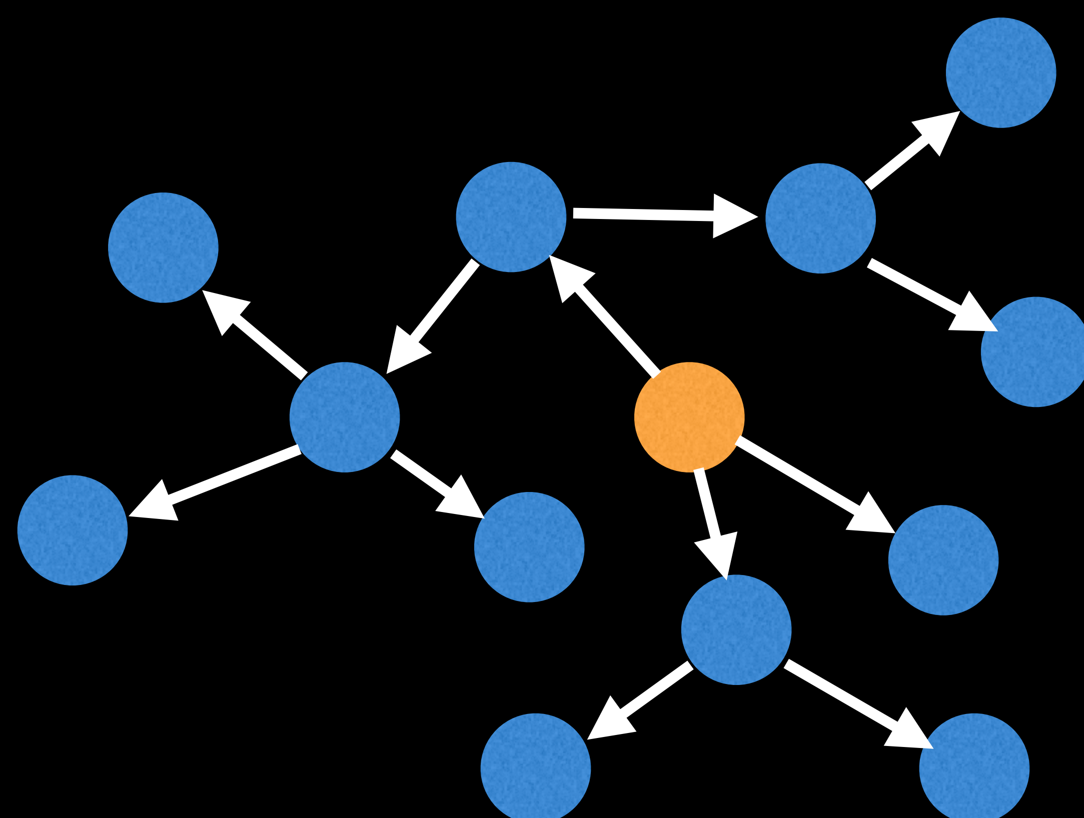
In practice, avoid storing a tree as an adjacency matrix! It’s a huge **waste of space** to use n^2 memory and only use $2(n-1)$ of the matrix cells.

Rooted Trees!

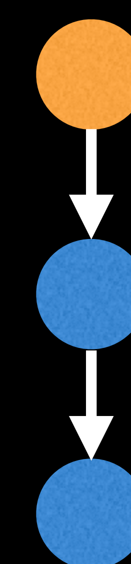
One of the more interesting types of trees is a **rooted tree** which is a tree with a designated **root node**.



Rooted tree



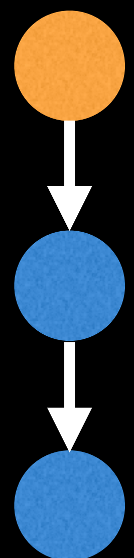
Rooted tree



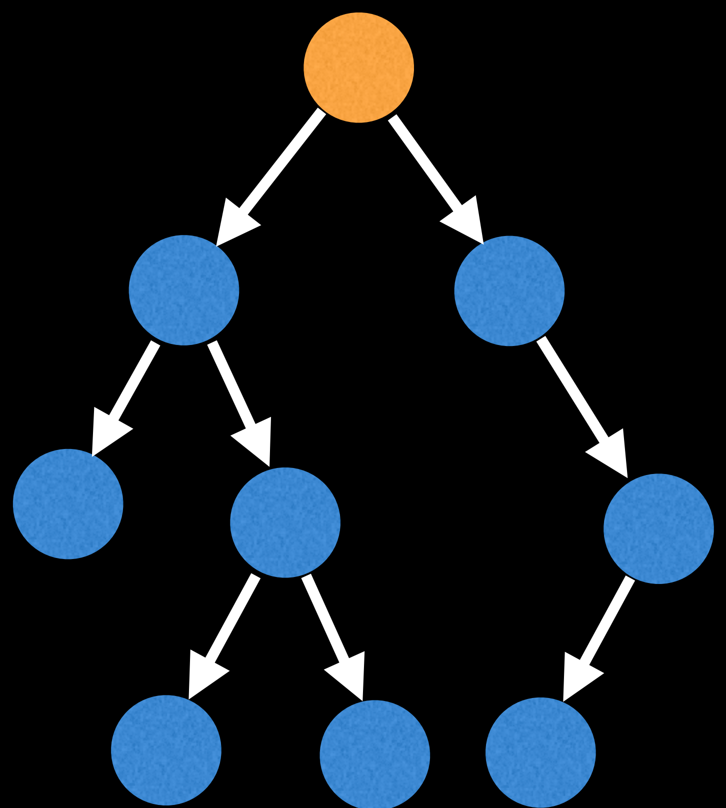
Rooted tree

Binary Tree (BT)

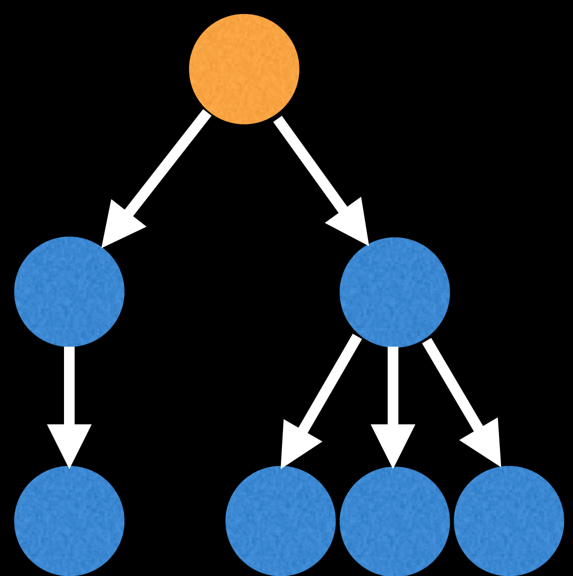
Related to rooted trees are **binary trees** which are trees for which every node has **at most two child nodes**.



Binary tree



Binary tree

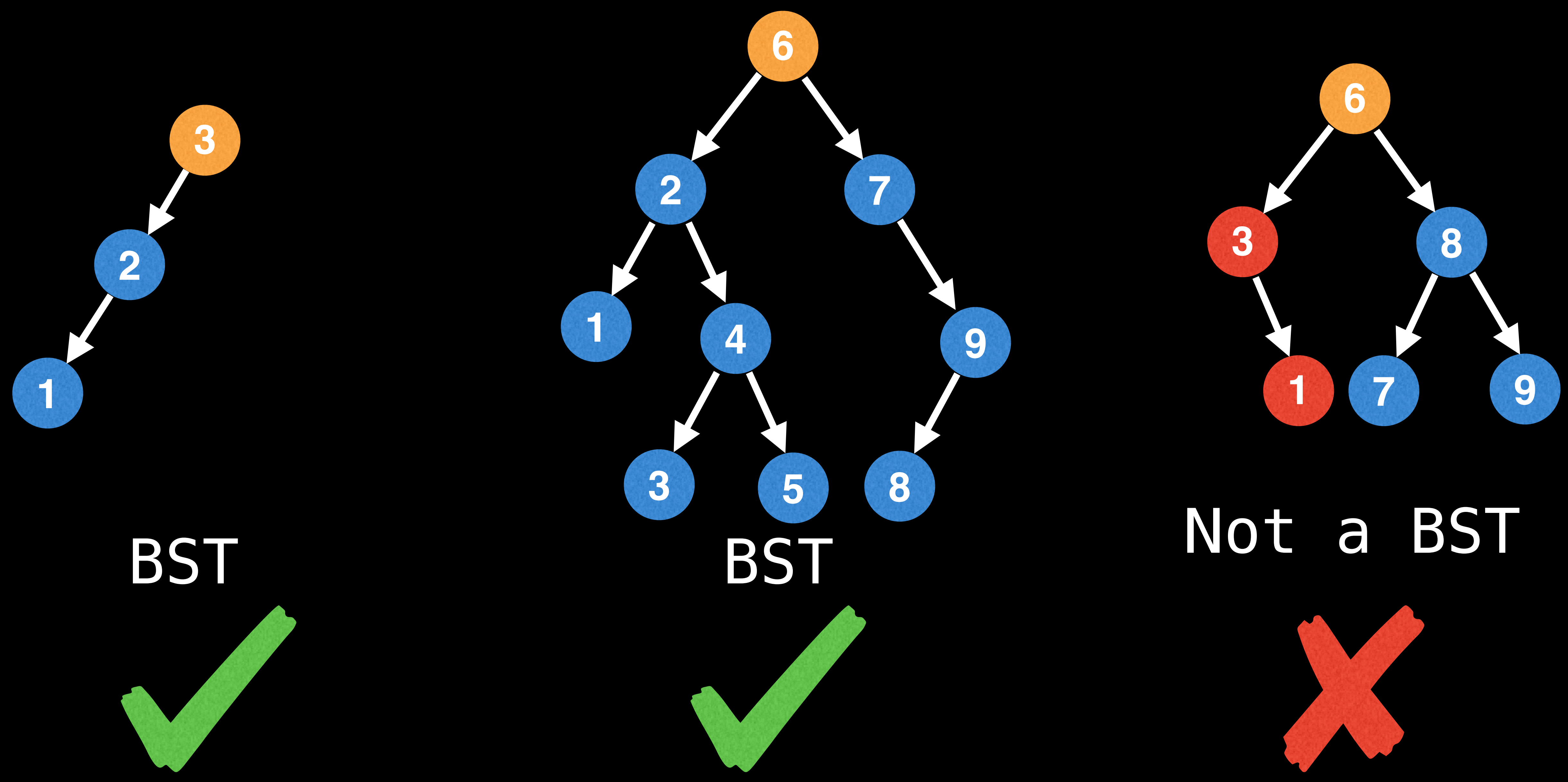


Not a binary tree

Binary Search Trees (BST)

Related to binary trees are **binary search trees** which are trees which satisfy the BST invariant which states that for every node x:

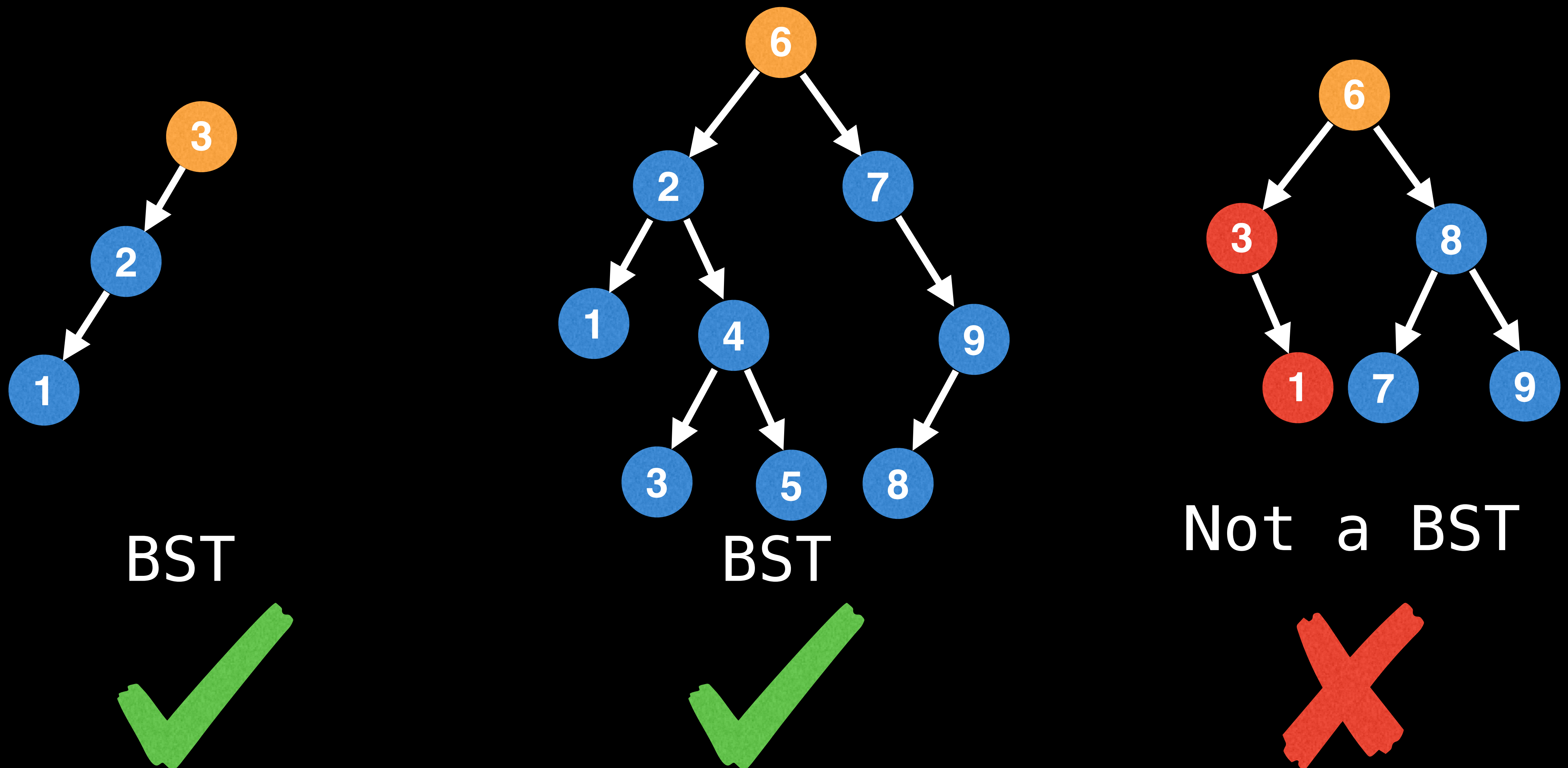
x.left.value ≤ x.value ≤ x.right.value



Binary Search Trees (BST)

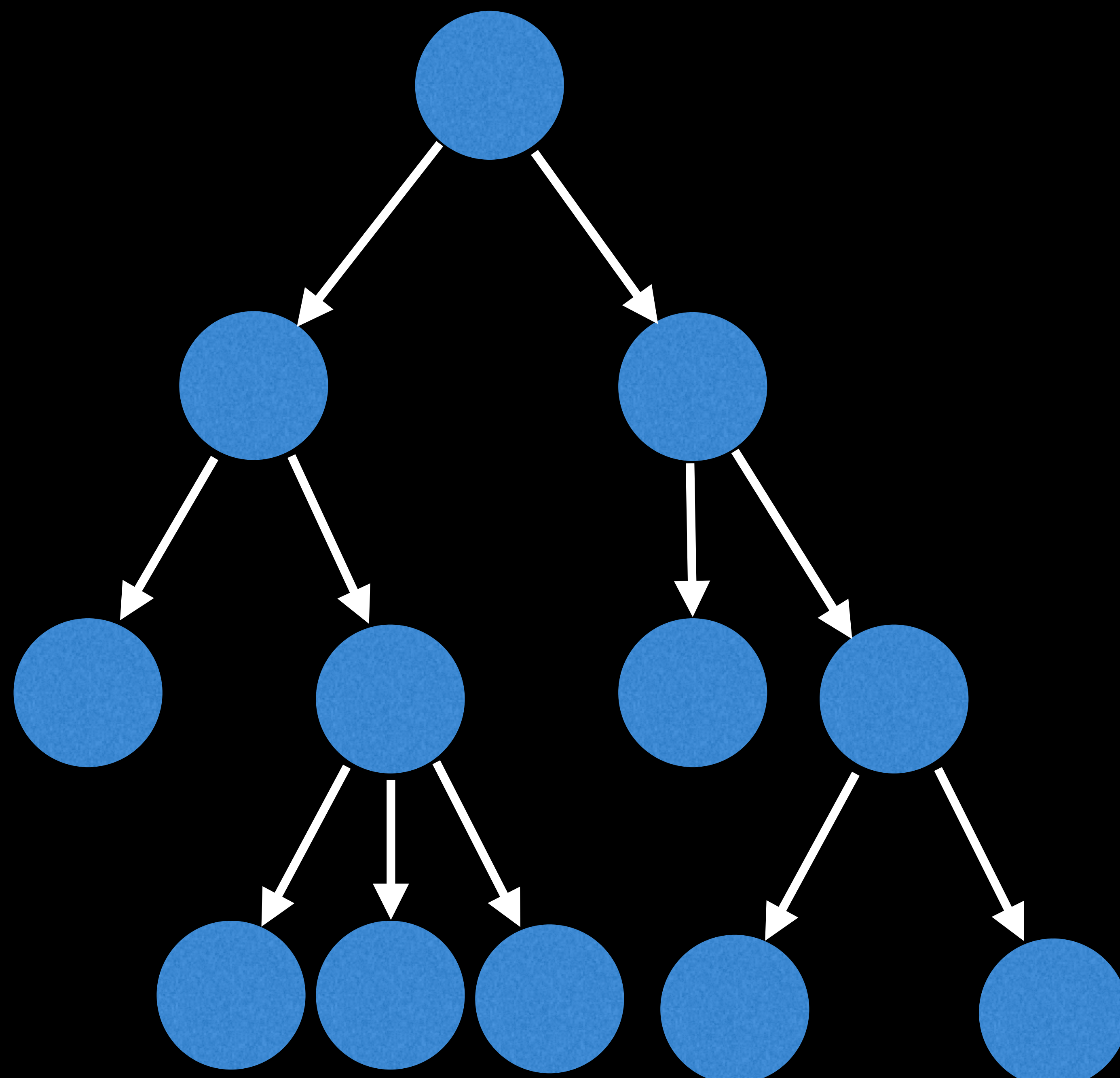
It’s often useful to **require uniqueness** on the node values in your tree. Change the invariant to be strictly < rather than ≤:

x.left.value < x.value < x.right.value



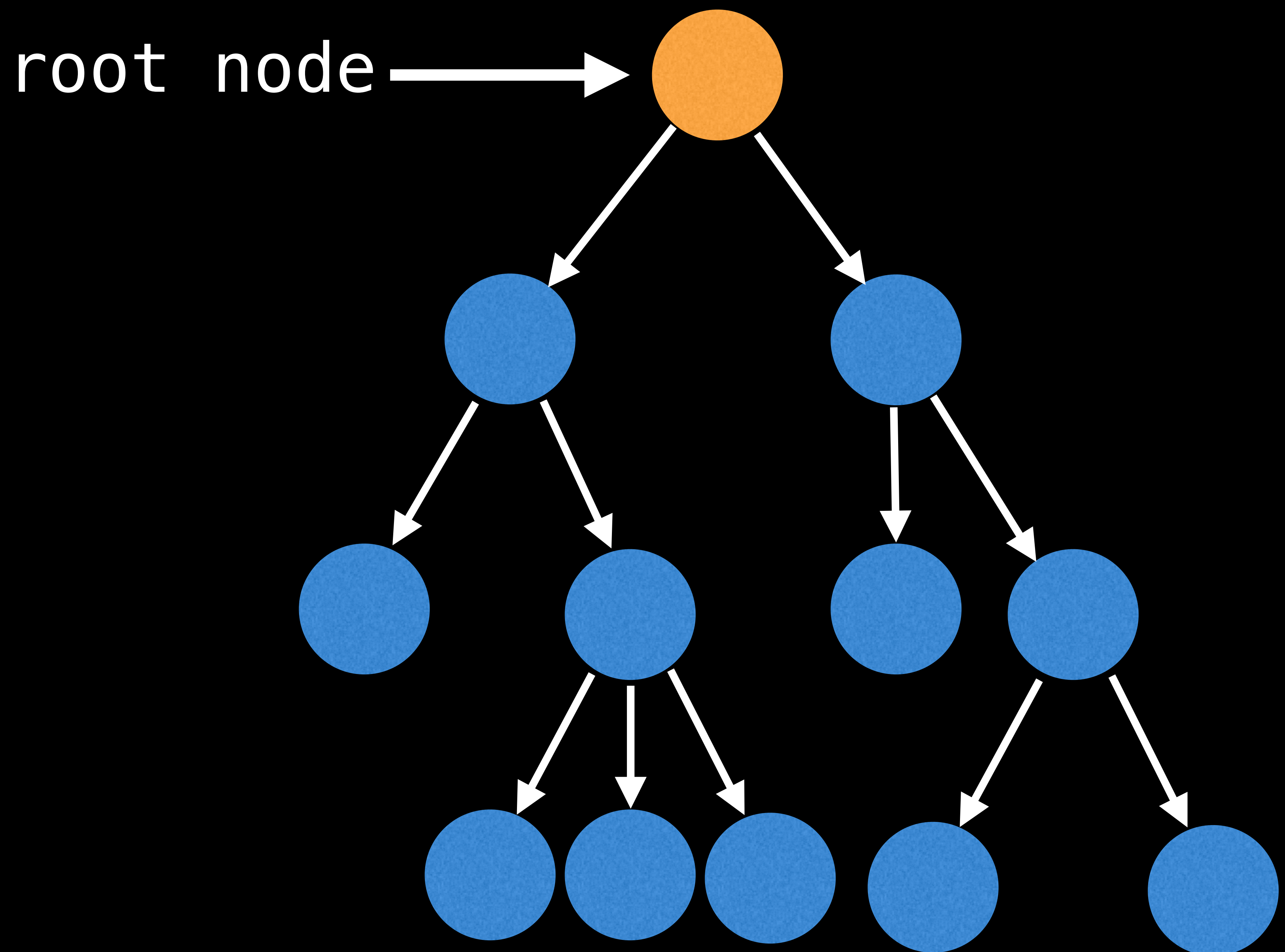
Storing rooted trees

Rooted trees are most naturally defined recursively in a top-down manner.



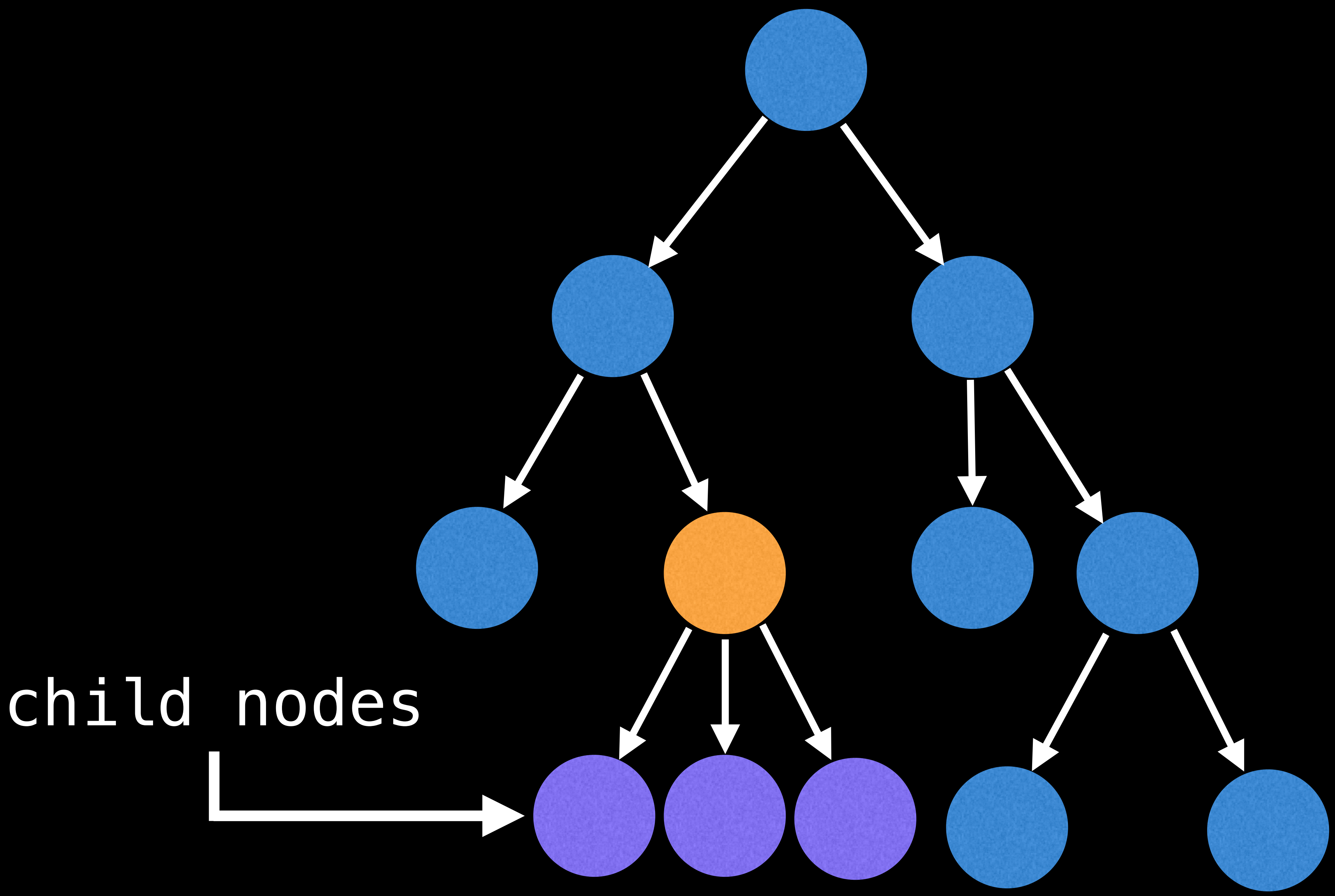
Storing rooted trees

In practice, you always maintain a pointer reference to the **root node** so that you can access the tree and its contents.



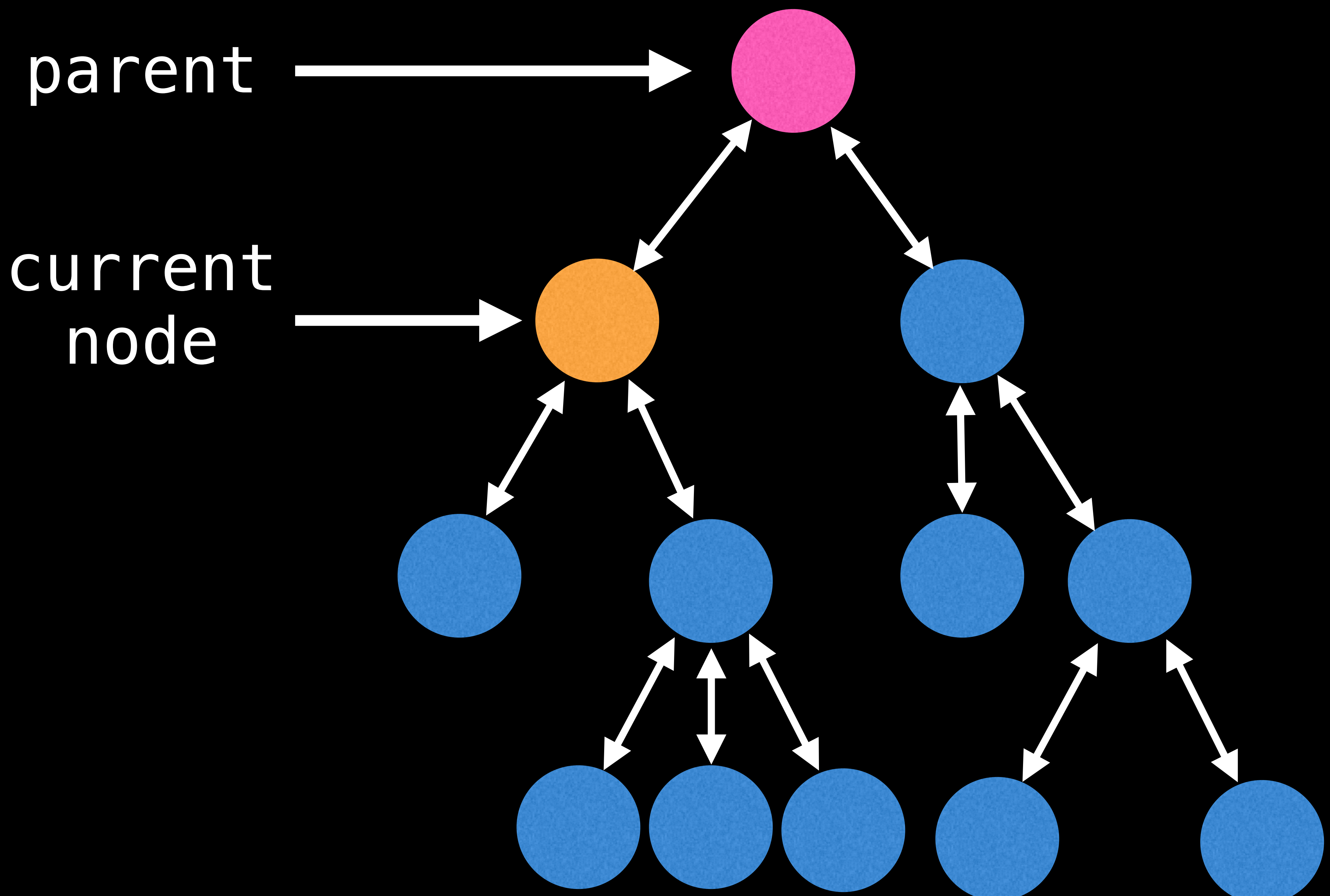
Storing rooted trees

Each node also has access to a list of all its **children**.



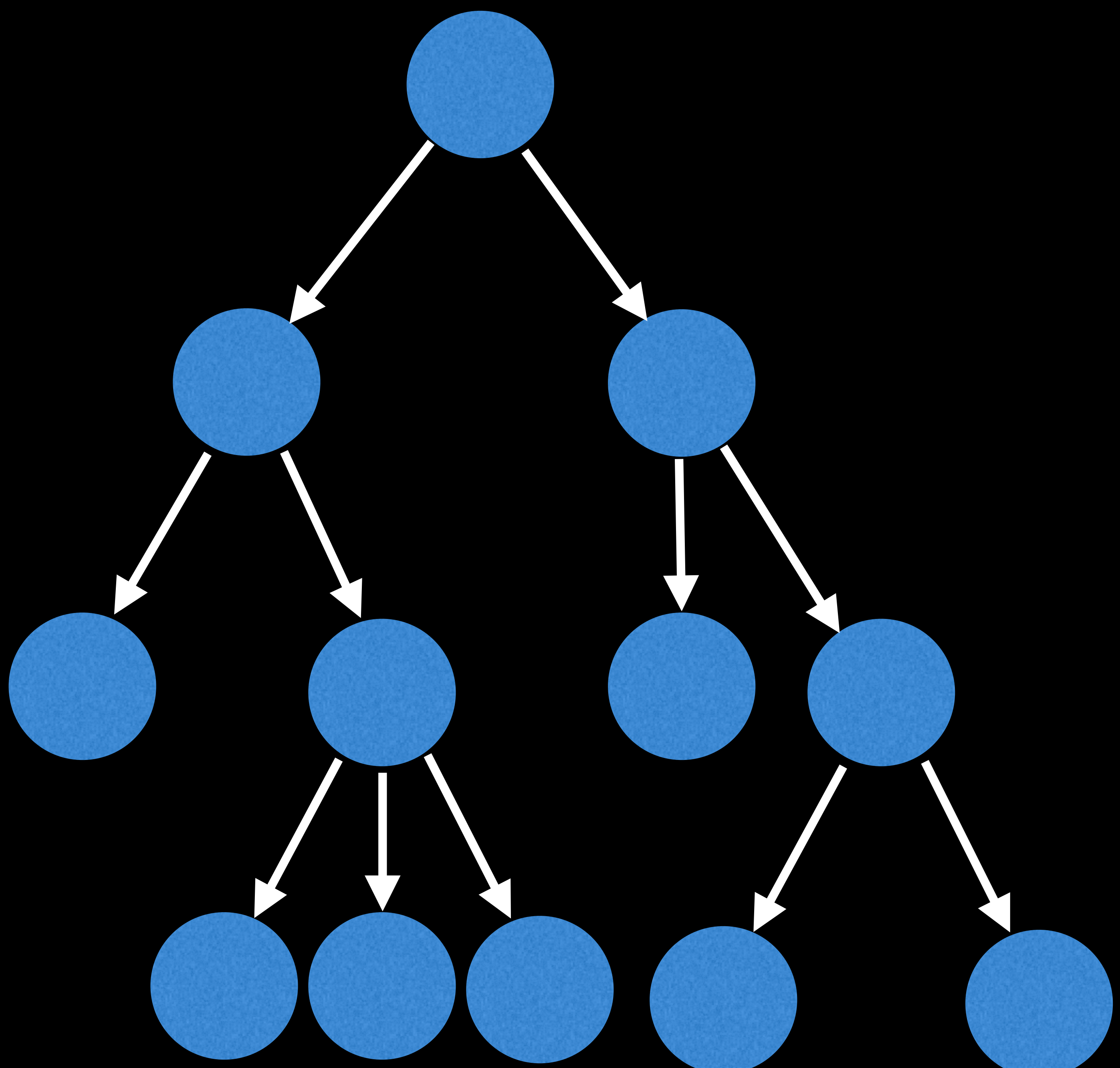
Storing rooted trees

Sometimes it's also useful to maintain a pointer to a node's **parent node** effectively making edges **bidirectional**.



Storing rooted trees

However, this isn't *usually* necessary because you can access a node's parent on a recursive function's callback.



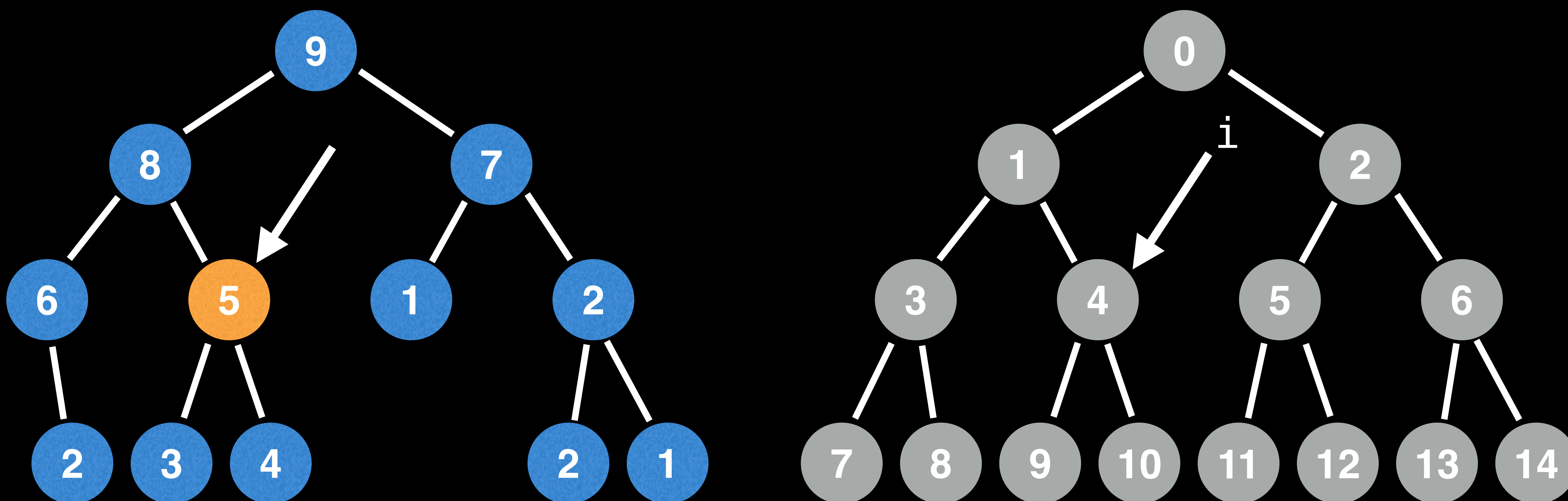
Storing rooted trees

If your tree is a **binary tree**, you can store it in a **flattened array**.

This trick also works for any n-ary tree

Storing rooted trees

In this flattened array representation, each node has an assigned index position based on where it is in the tree.

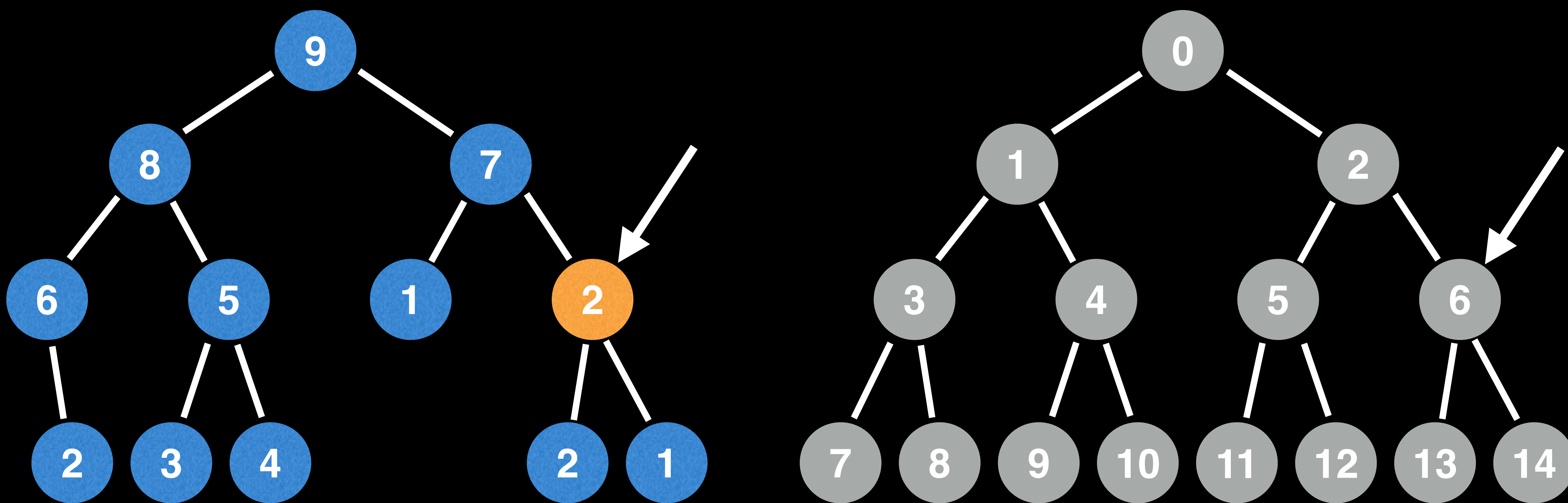


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	∅	2	3	4	∅	∅	2	1

This trick also works for any n-ary tree

Storing rooted trees

In this flattened array representation, each node has an assigned index position based on where it is in the tree.

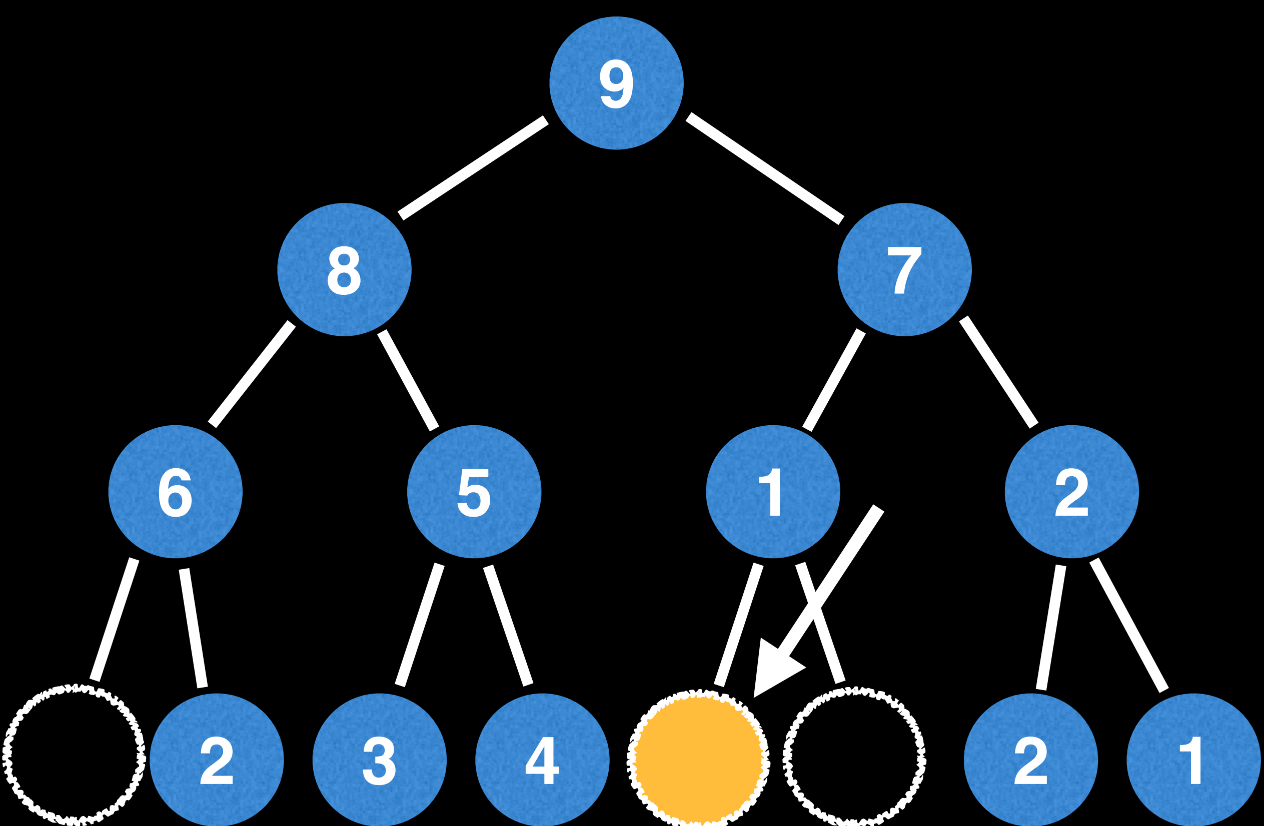


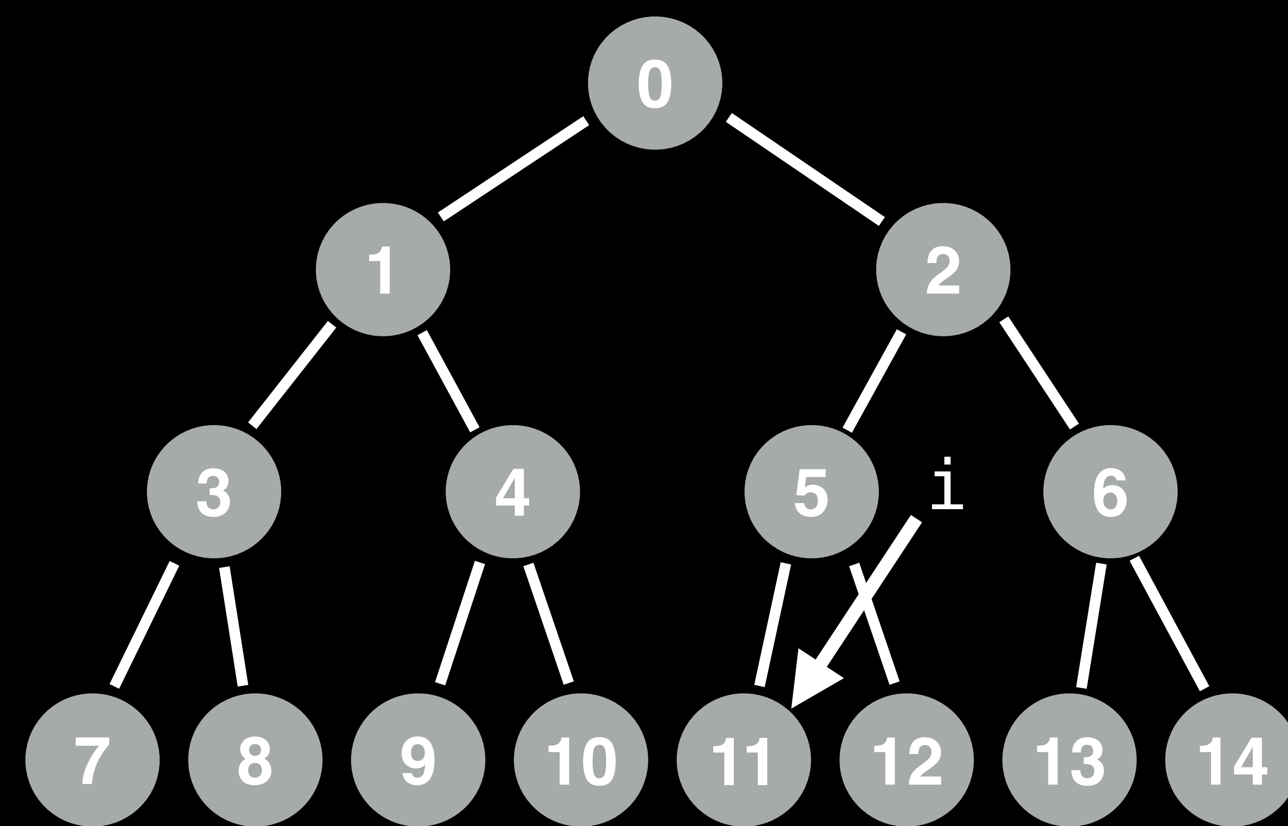
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	∅	2	3	4	∅	∅	2	1

This trick also works for any n-ary tree

Storing rooted trees

Even nodes which aren't currently present have an index because they can be mapped back to a unique position in the "index tree" (gray tree).





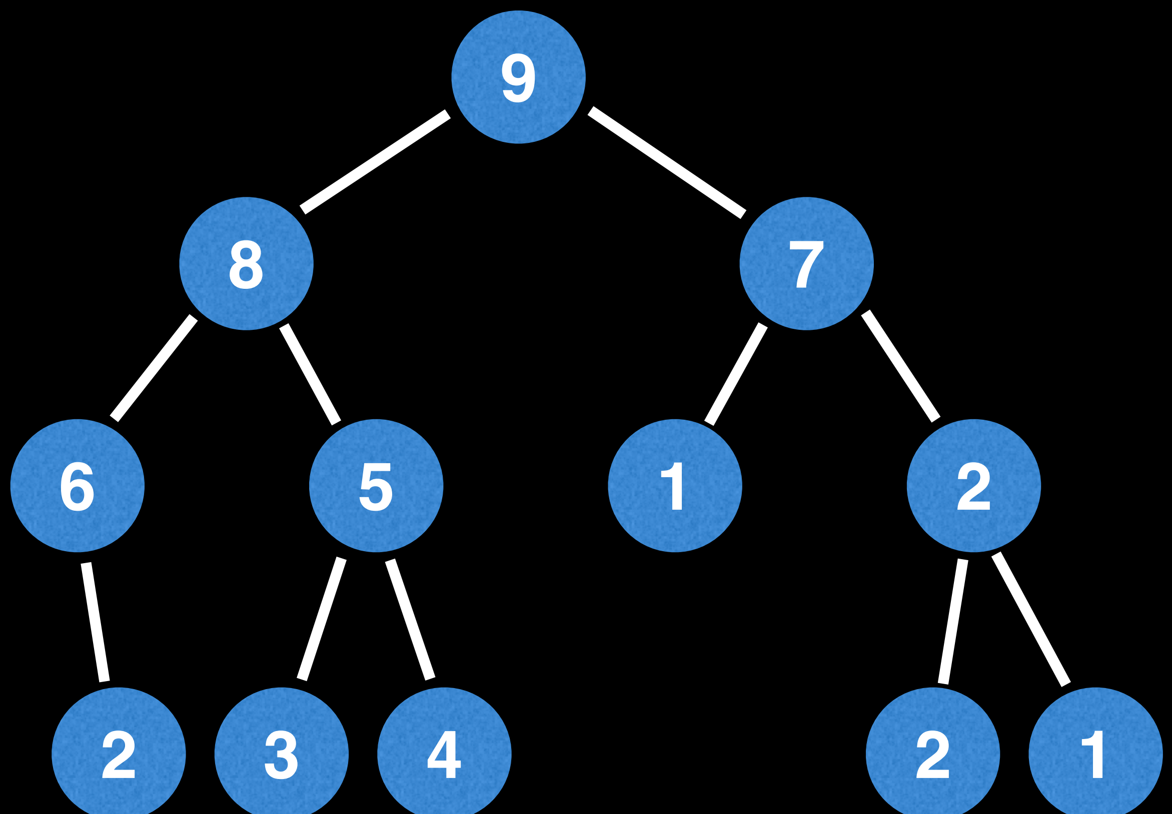
index tree

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	∅	2	3	4	∅	∅	2	1

This trick also works for any n-ary tree

Storing rooted trees

The root node is always at index 0 and the children of the current node *i* are accessed relative to position *i*.



Let *i* be the index of the current node

left node: 2*i + 1
right node: 2*i + 2

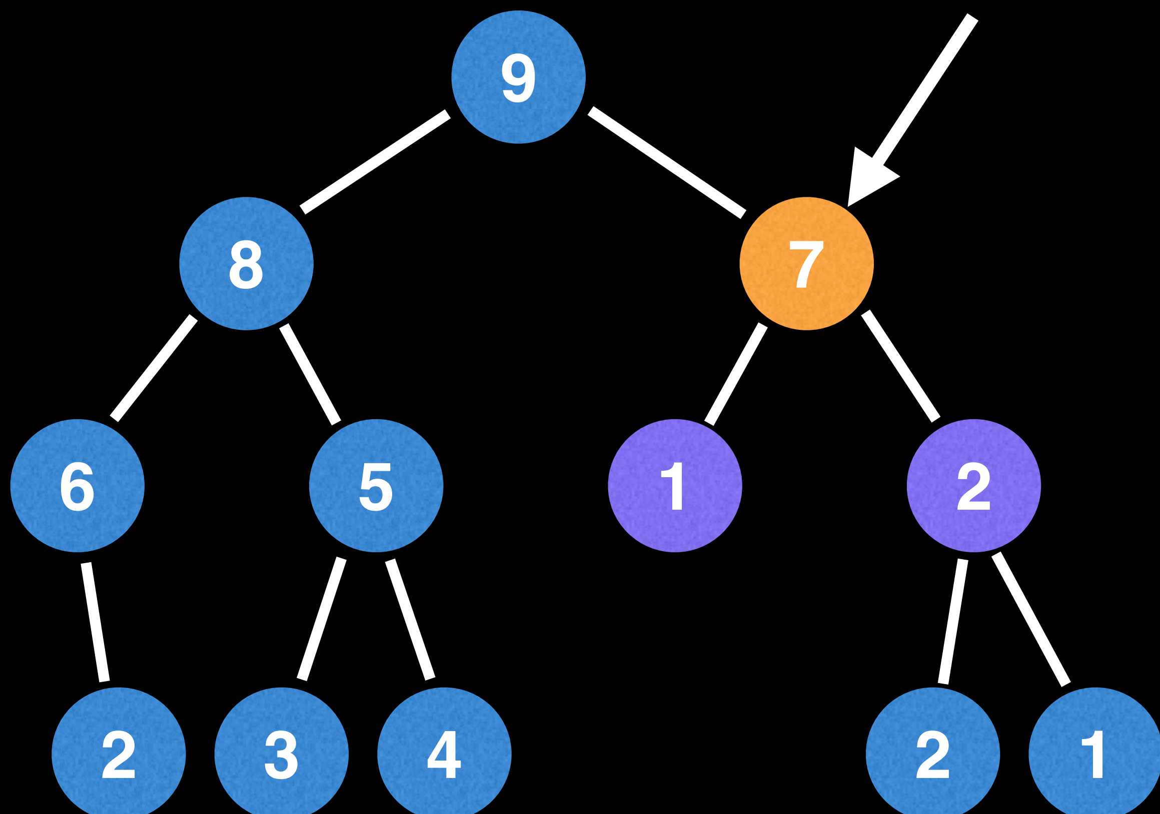
Reciprocally, the parent of node i is: floor((i-1)/2)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	∅	2	3	4	∅	∅	2	1

This trick also works for any n-ary tree

Storing rooted trees

The root node is always at index 0 and the children of the current node *i* are accessed relative to position *i*.



Let *i* be the index of the current node

left node: 2*i + 1
right node: 2*i + 2

Reciprocally, the parent of node i is: floor((i-1)/2)

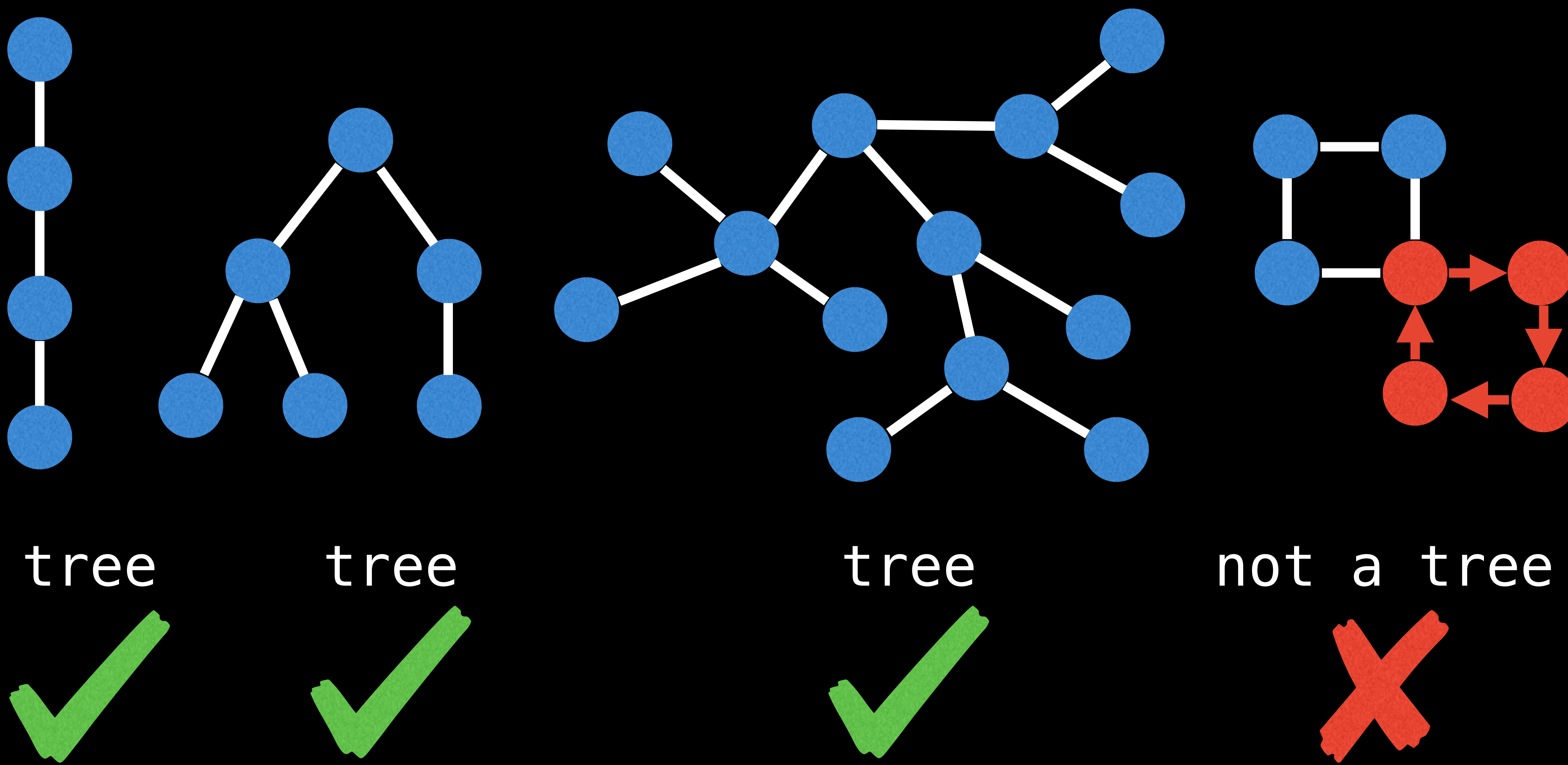
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	∅	2	3	4	∅	∅	2	1

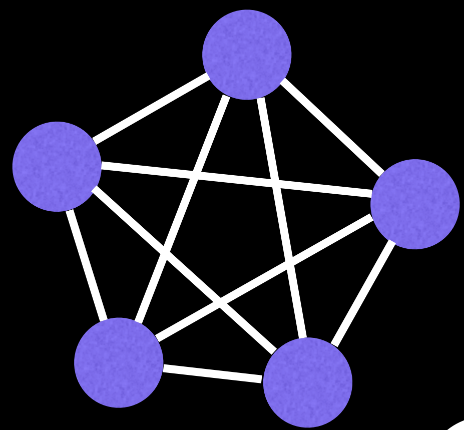
This trick also works for any n-ary tree

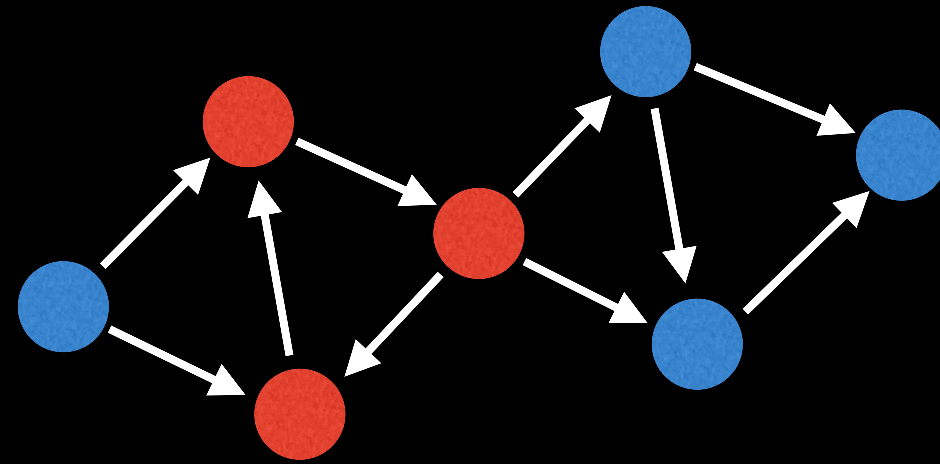
Next Video: beginner tree algorithms

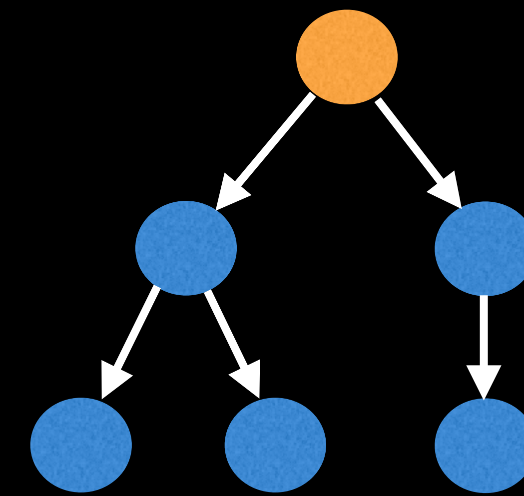
Introduction to Trees

A **tree** is a connected, undirected graph with **no cycles**.

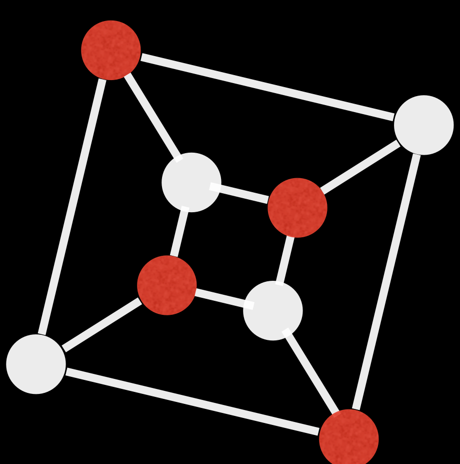


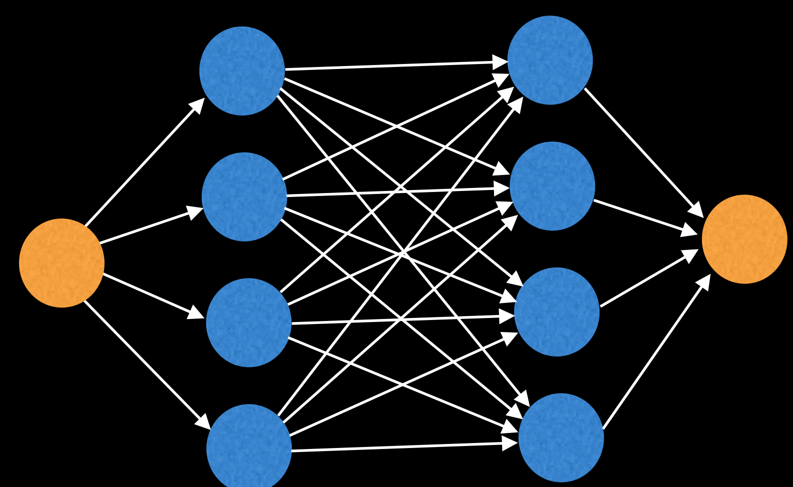


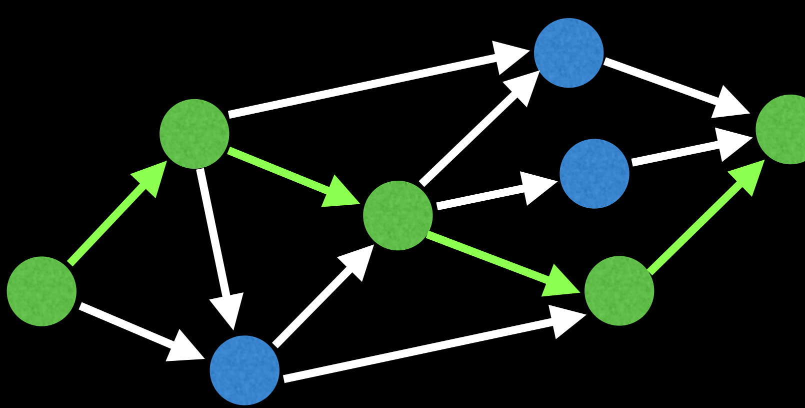




Graph Theory Video Series





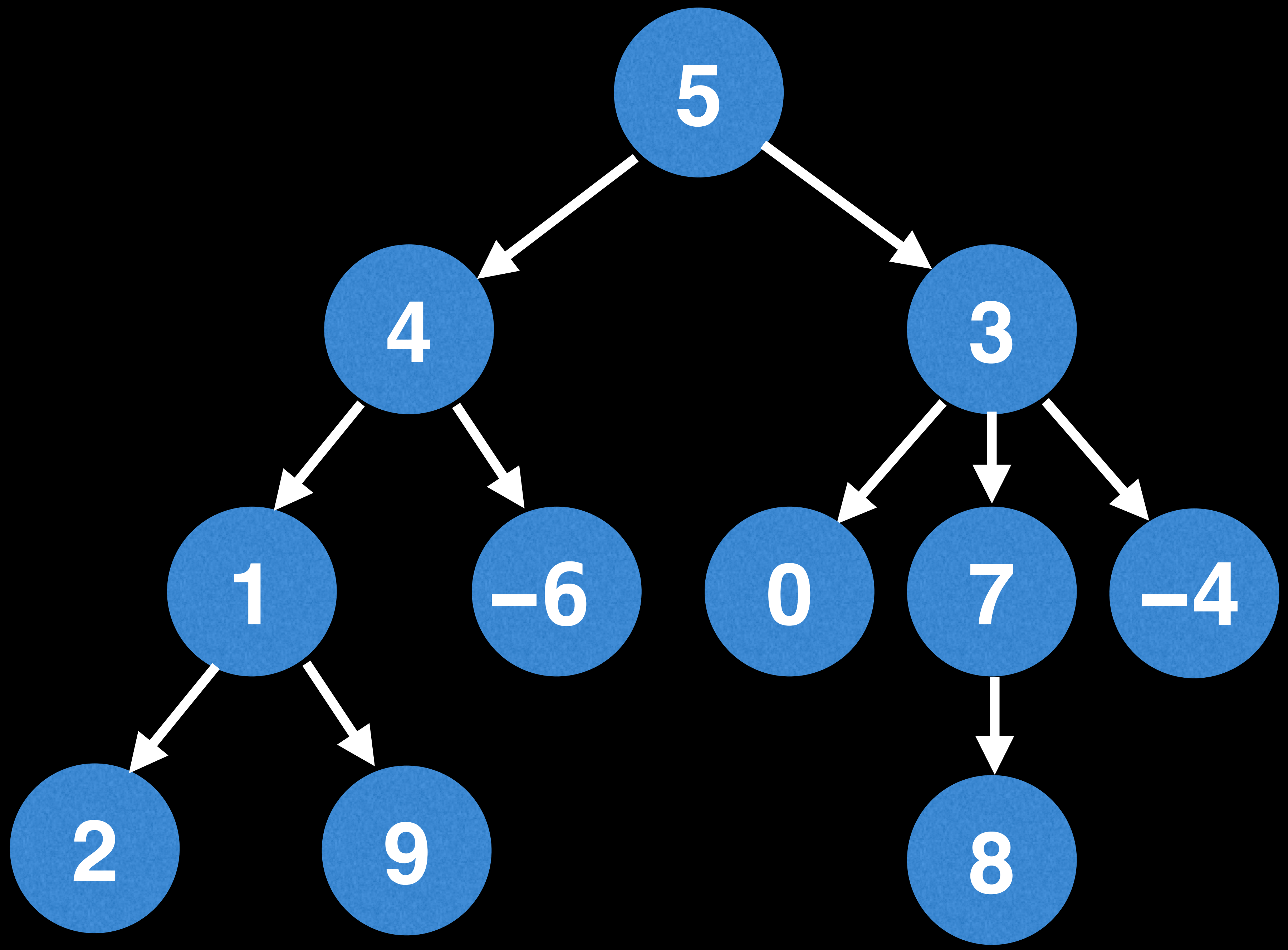


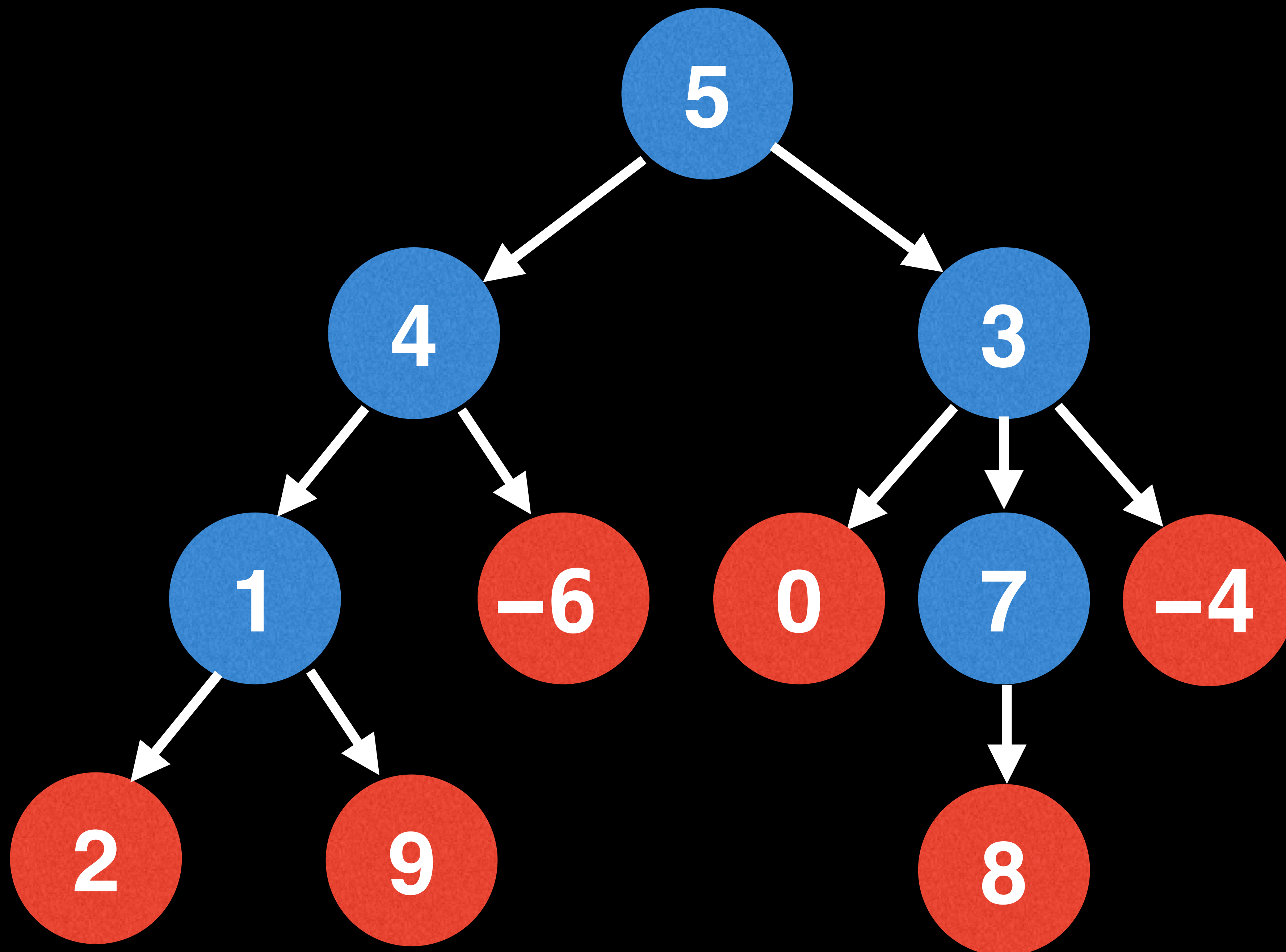
Beginner tree algorithms

William Fiset

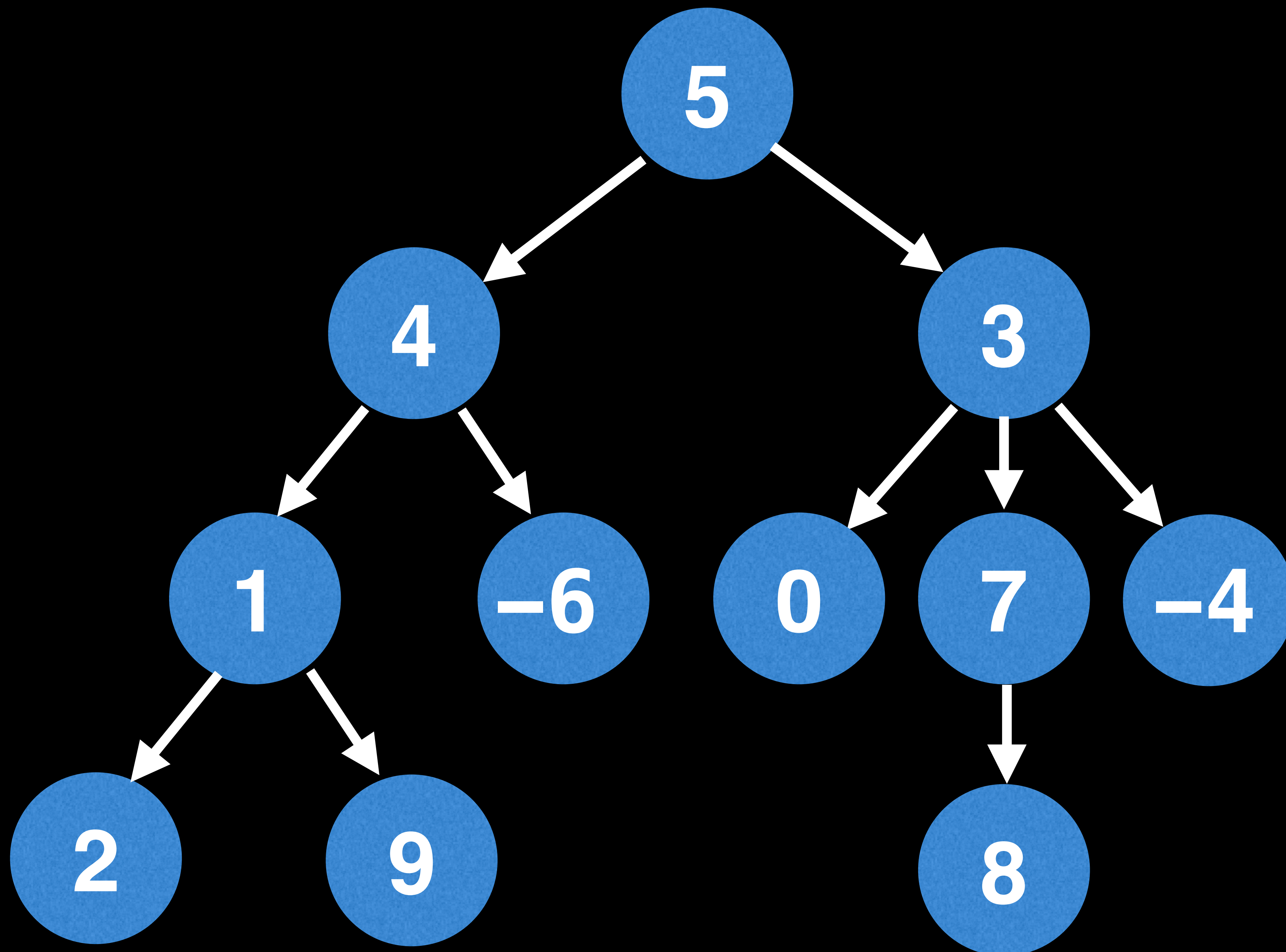
Problem 1: leaf node sum

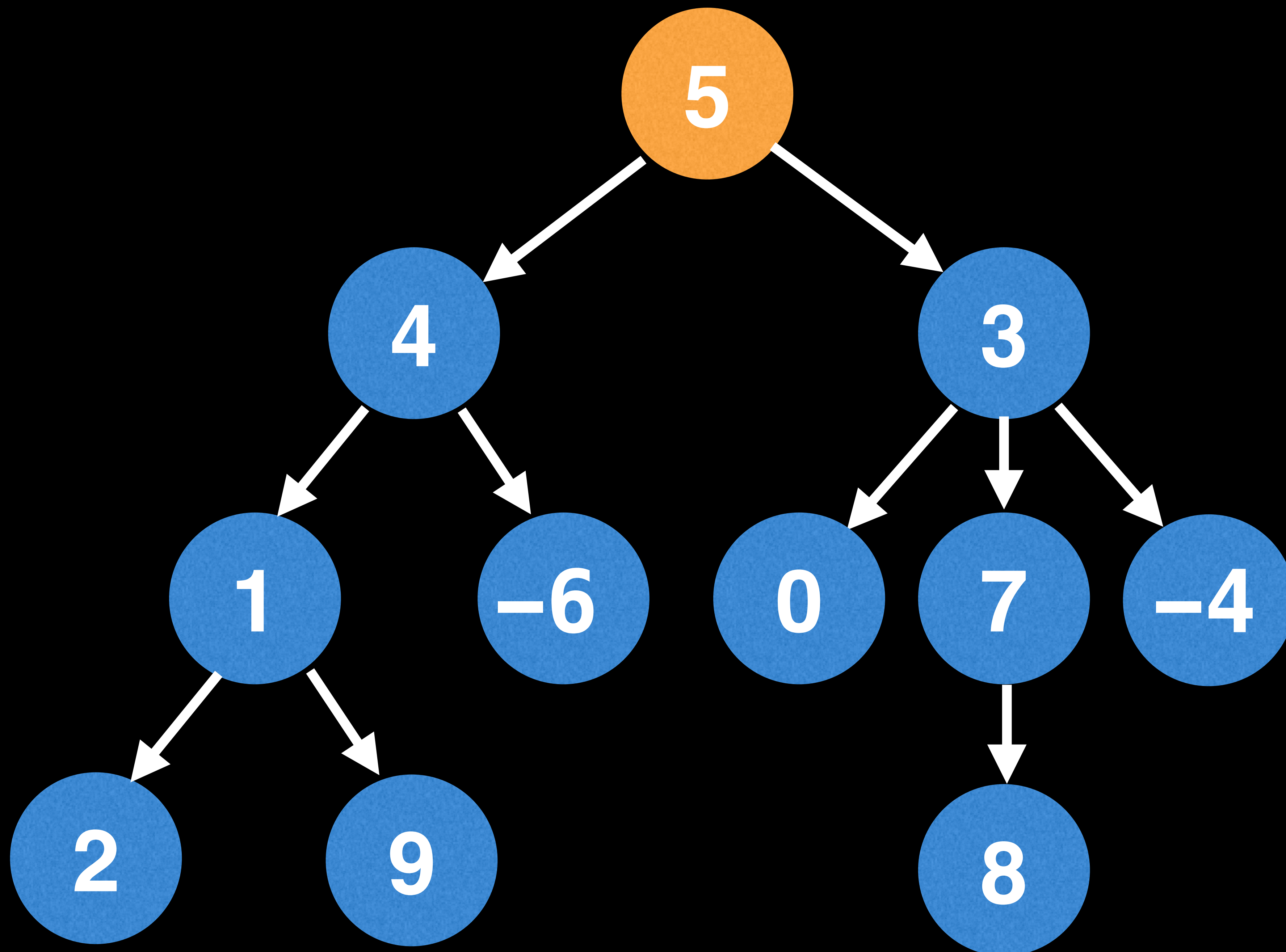
What is the sum of all the leaf node values in a tree?


5
4
3
1
−6
0
7
−4
2
9
8

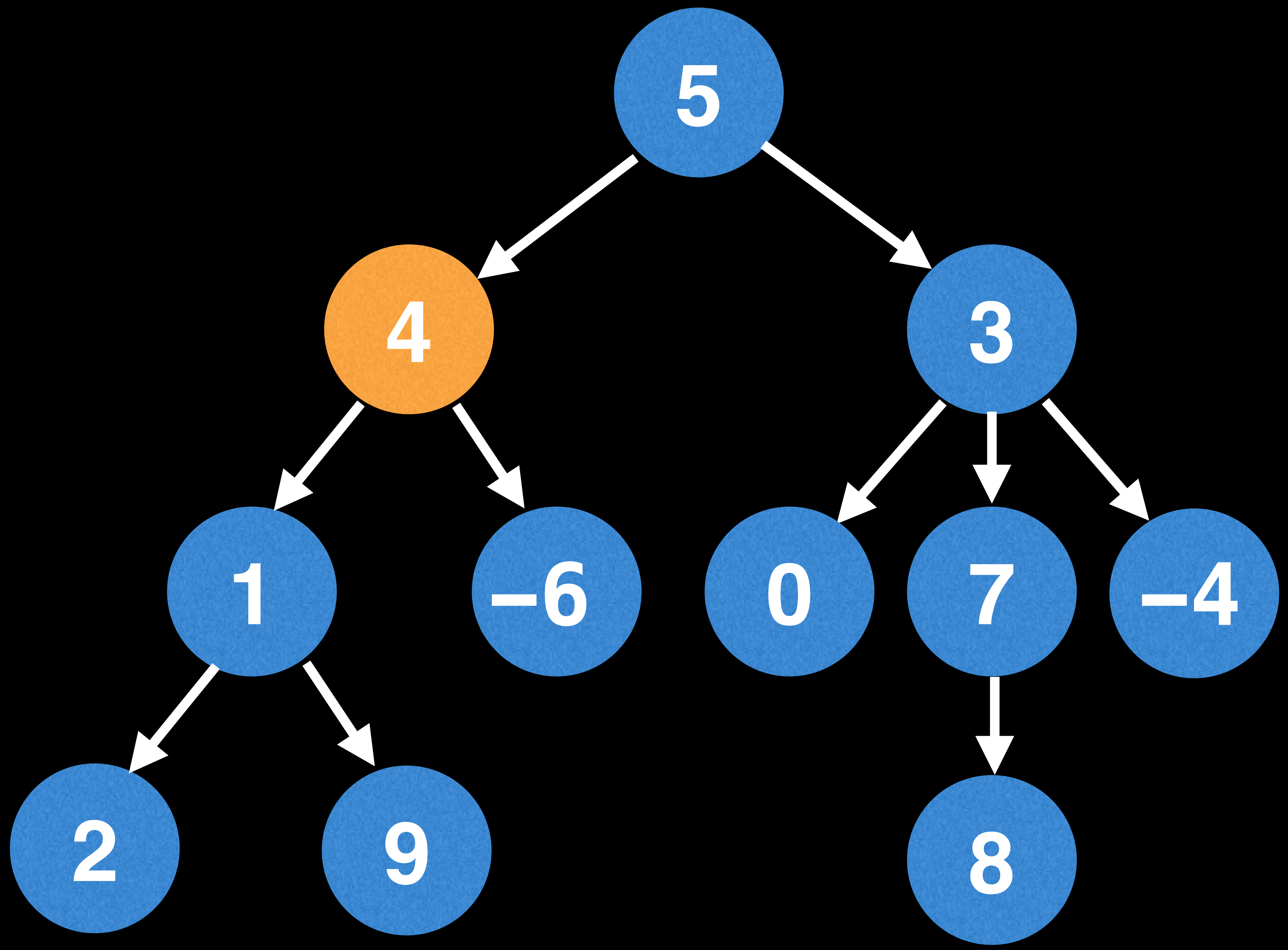


$$2 + 9 - 6 + 0 + 8 - 4 = 9$$

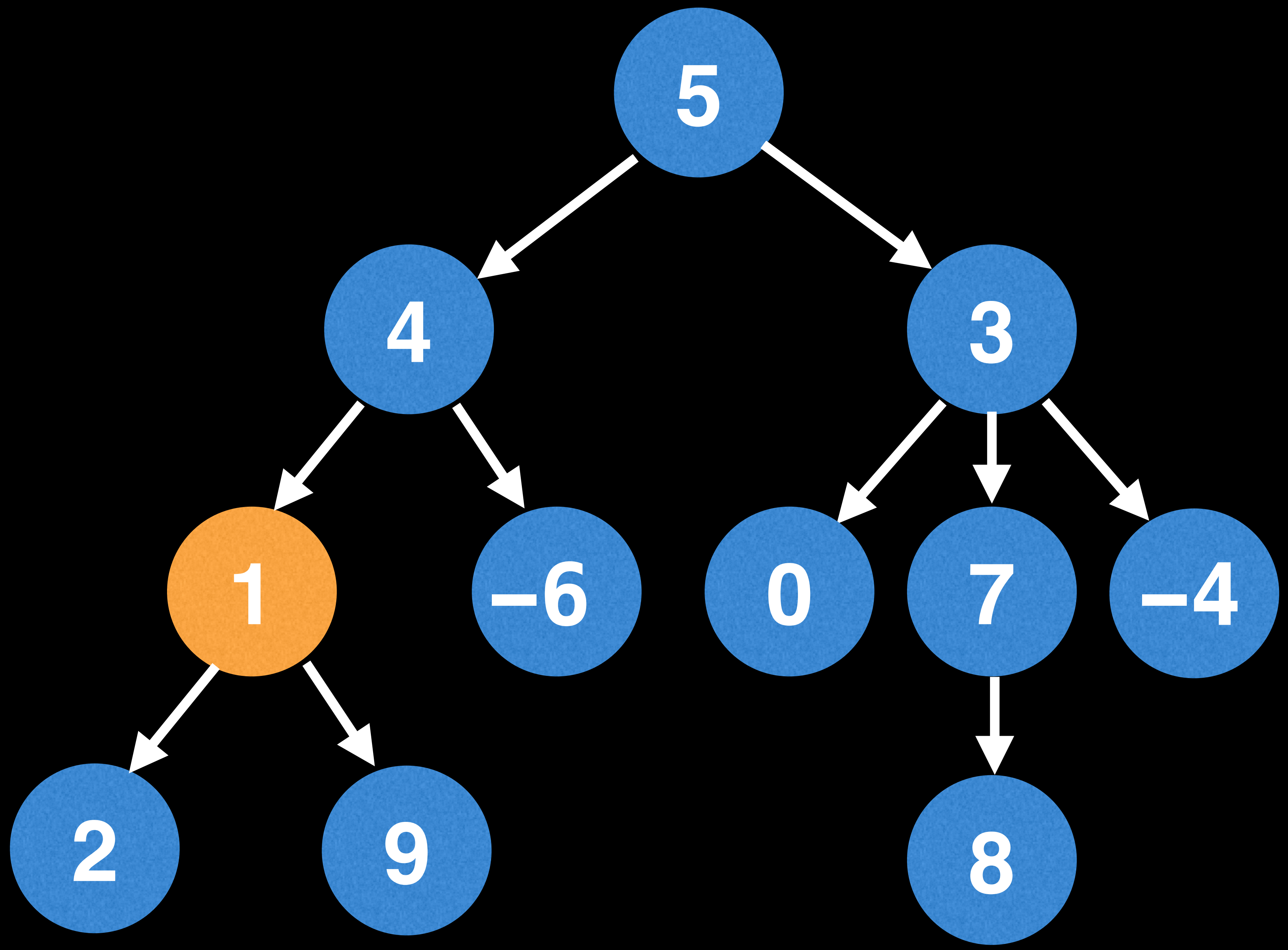

5
4
3
1
–6
0
7
–4
2
9
8

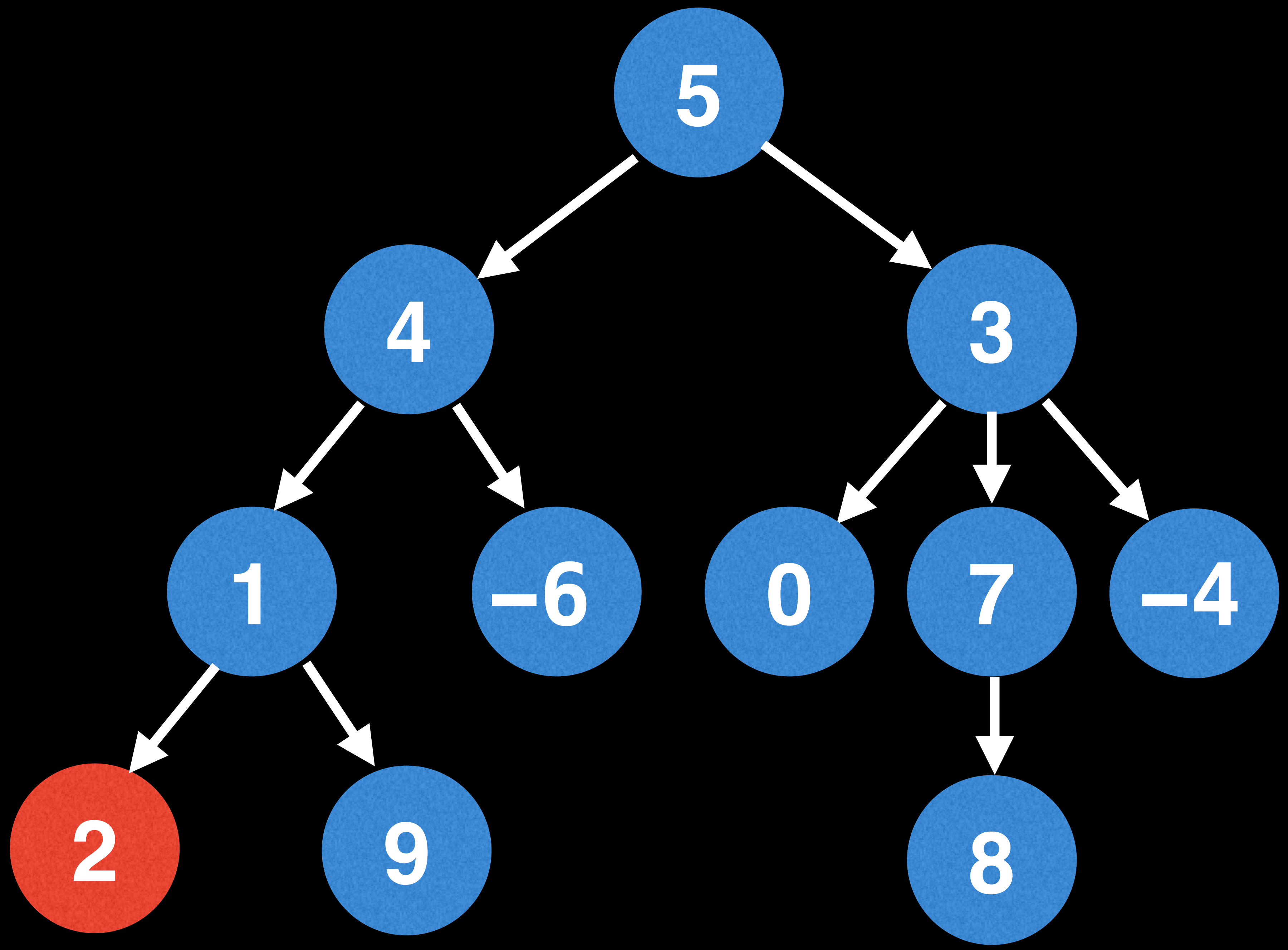


When dealing with rooted trees you begin with having a reference to the root node as a starting point for most algorithms.

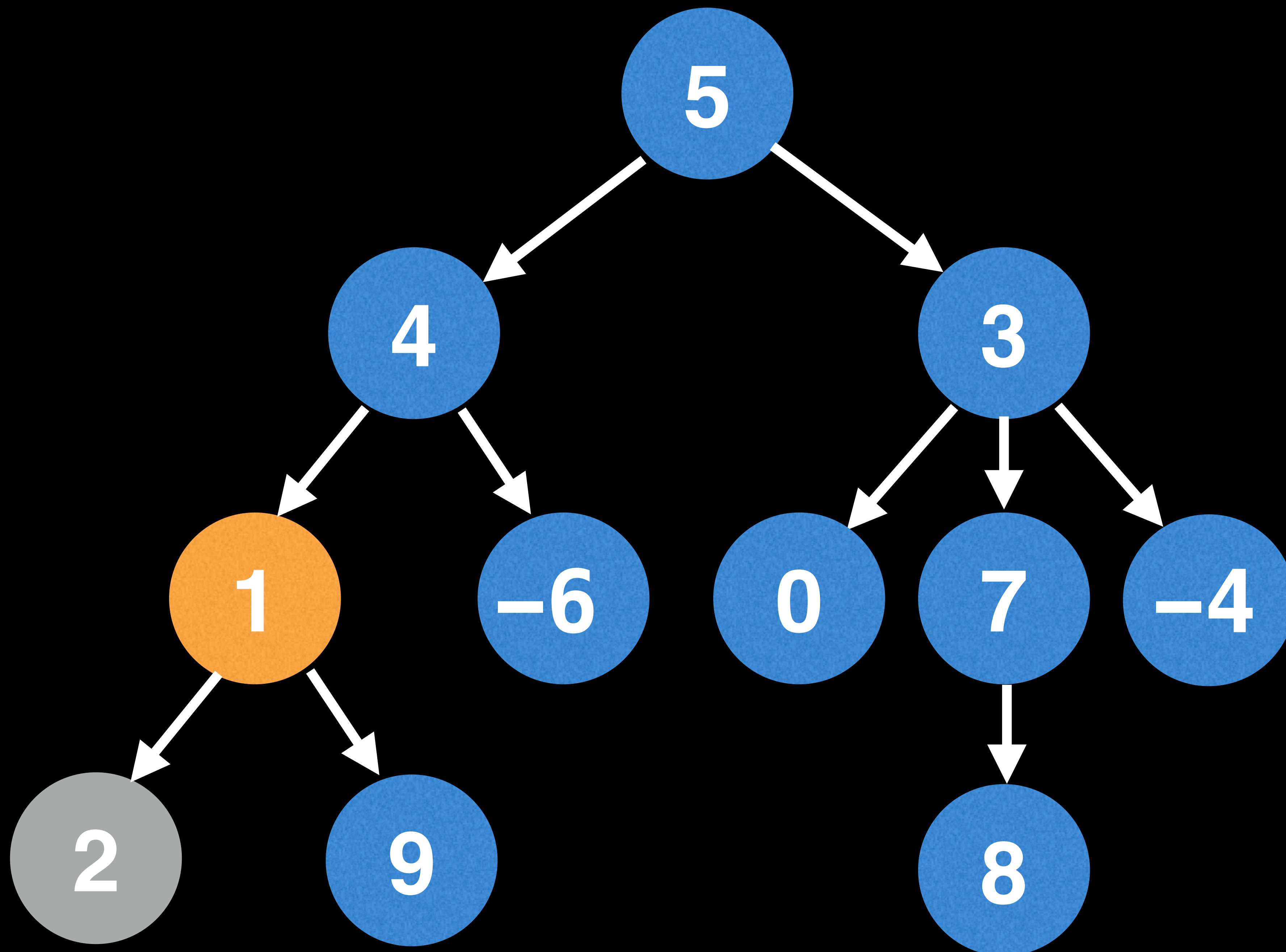


5
4
3
1
−6
0
7
−4
2
9
8

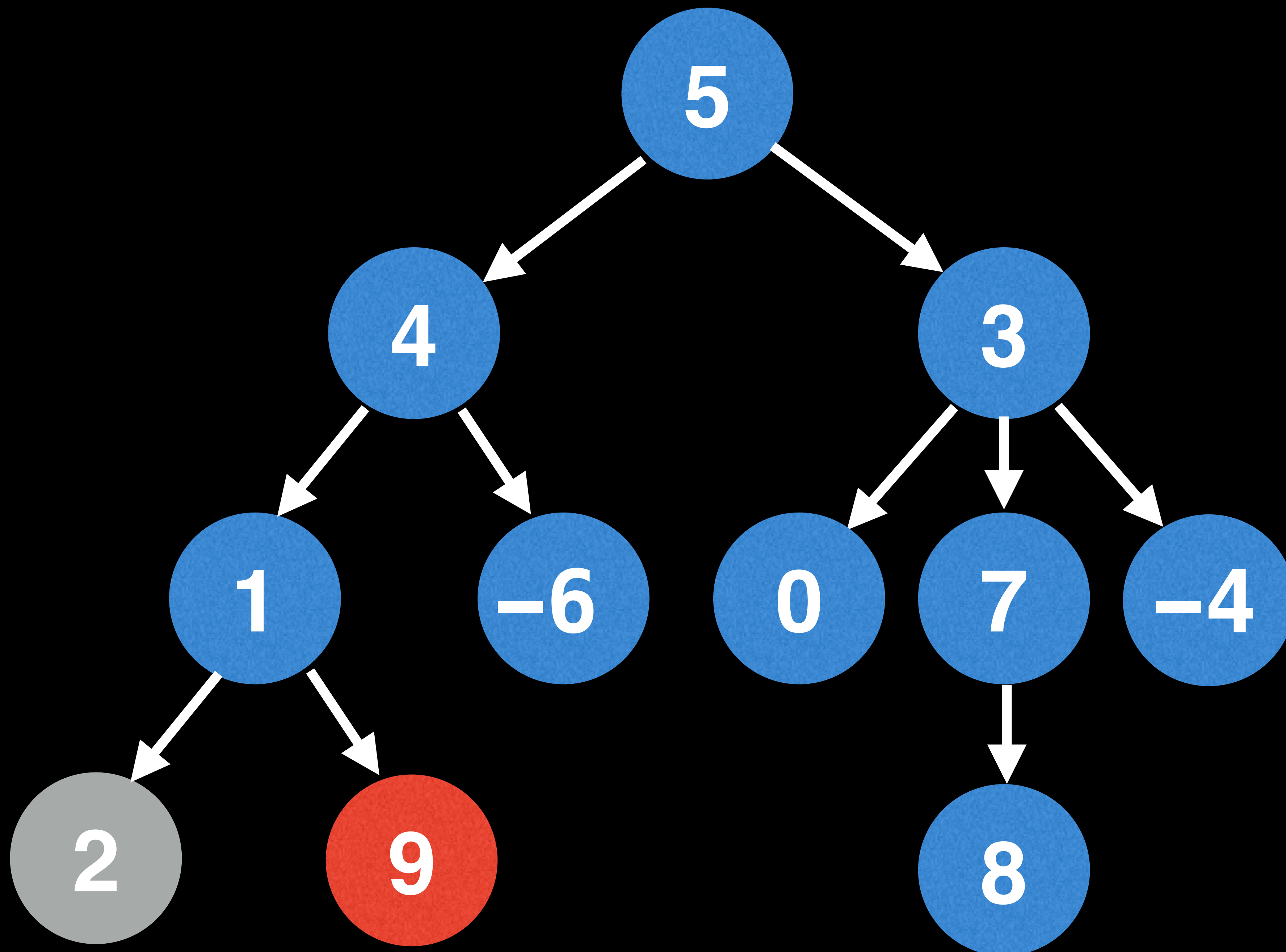

5
4
3
1
-6
0
7
-4
2
9
8



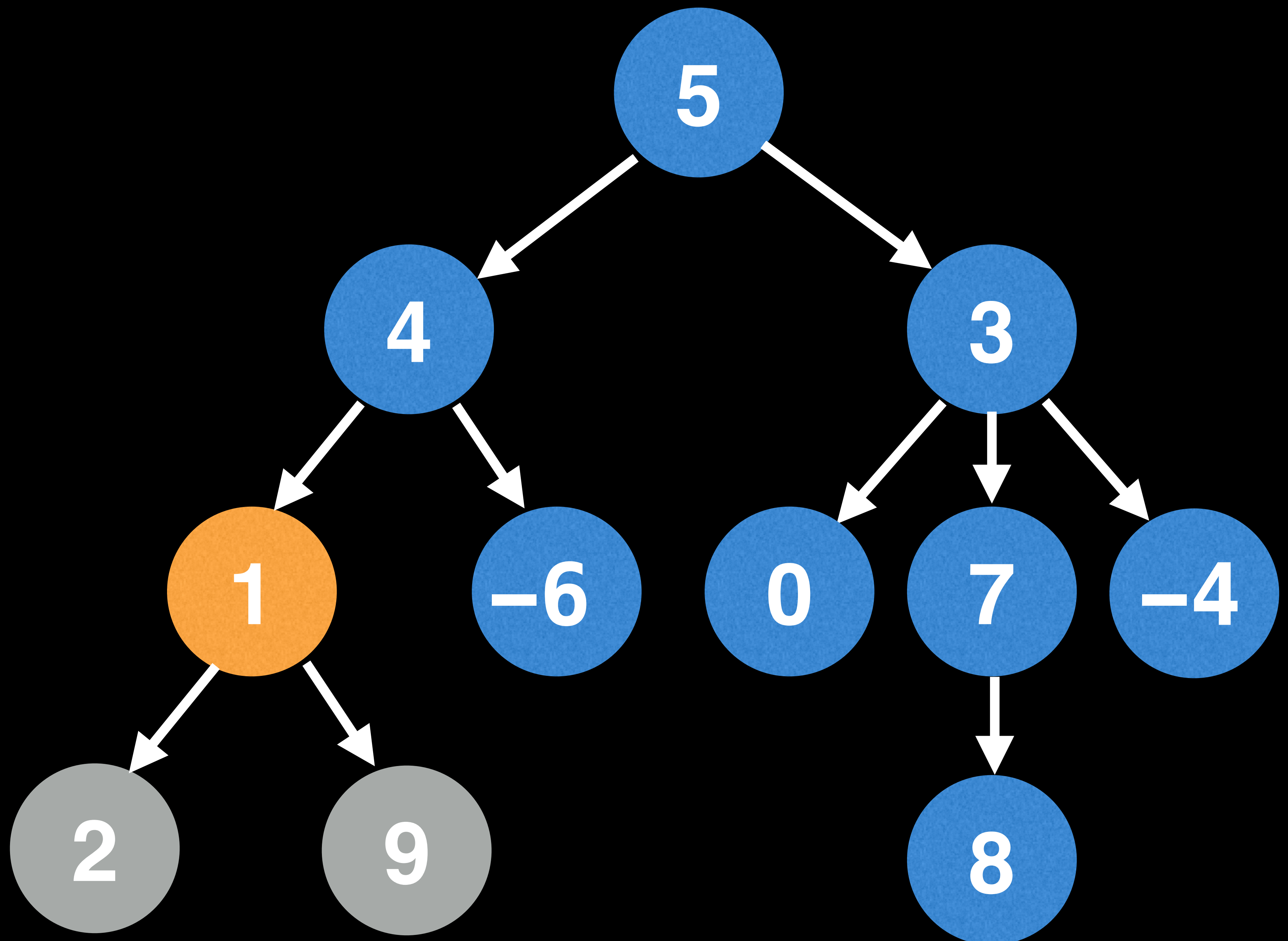
5
4
3
1
−6
0
7
−4
2
9
8



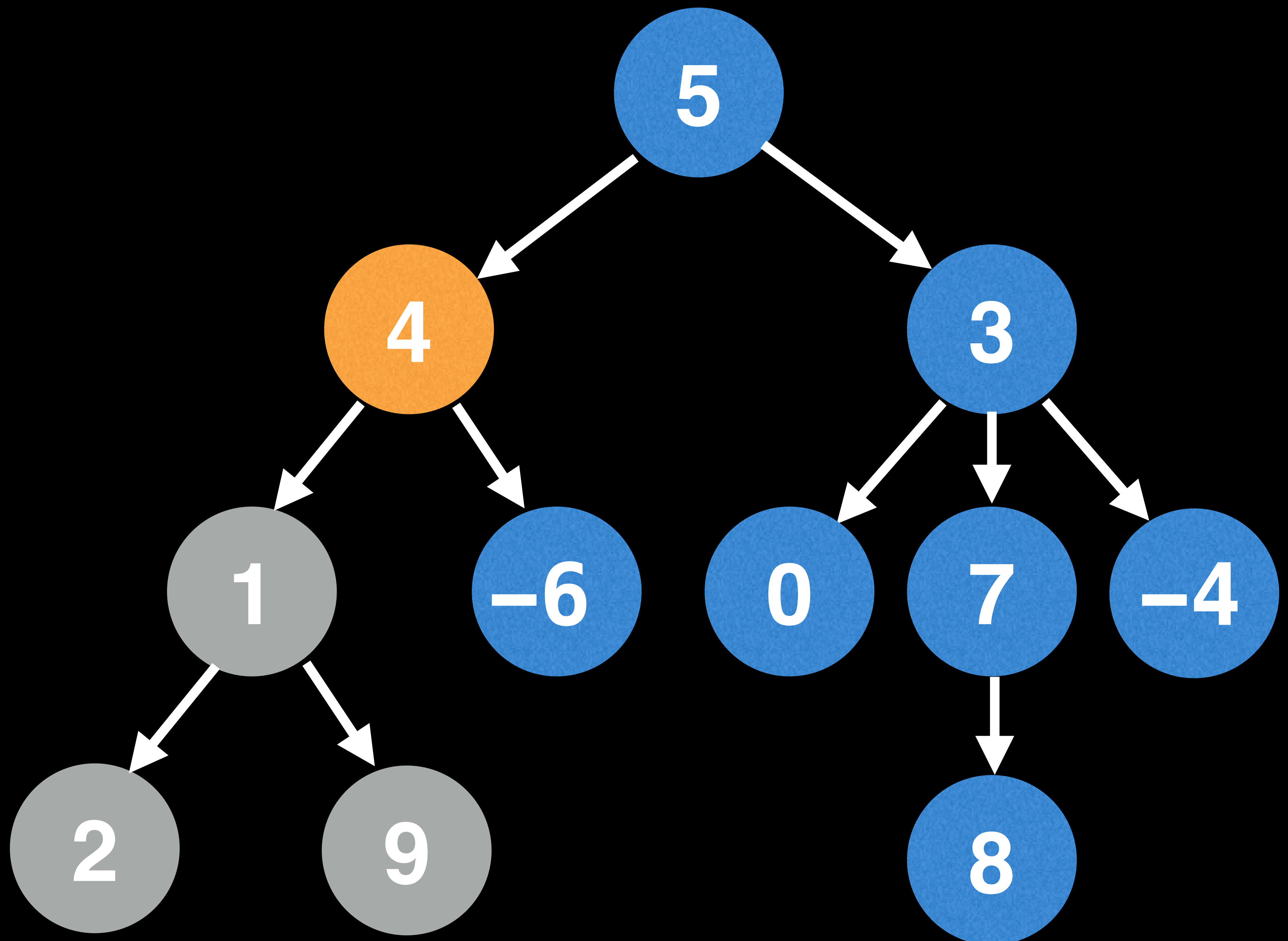
2



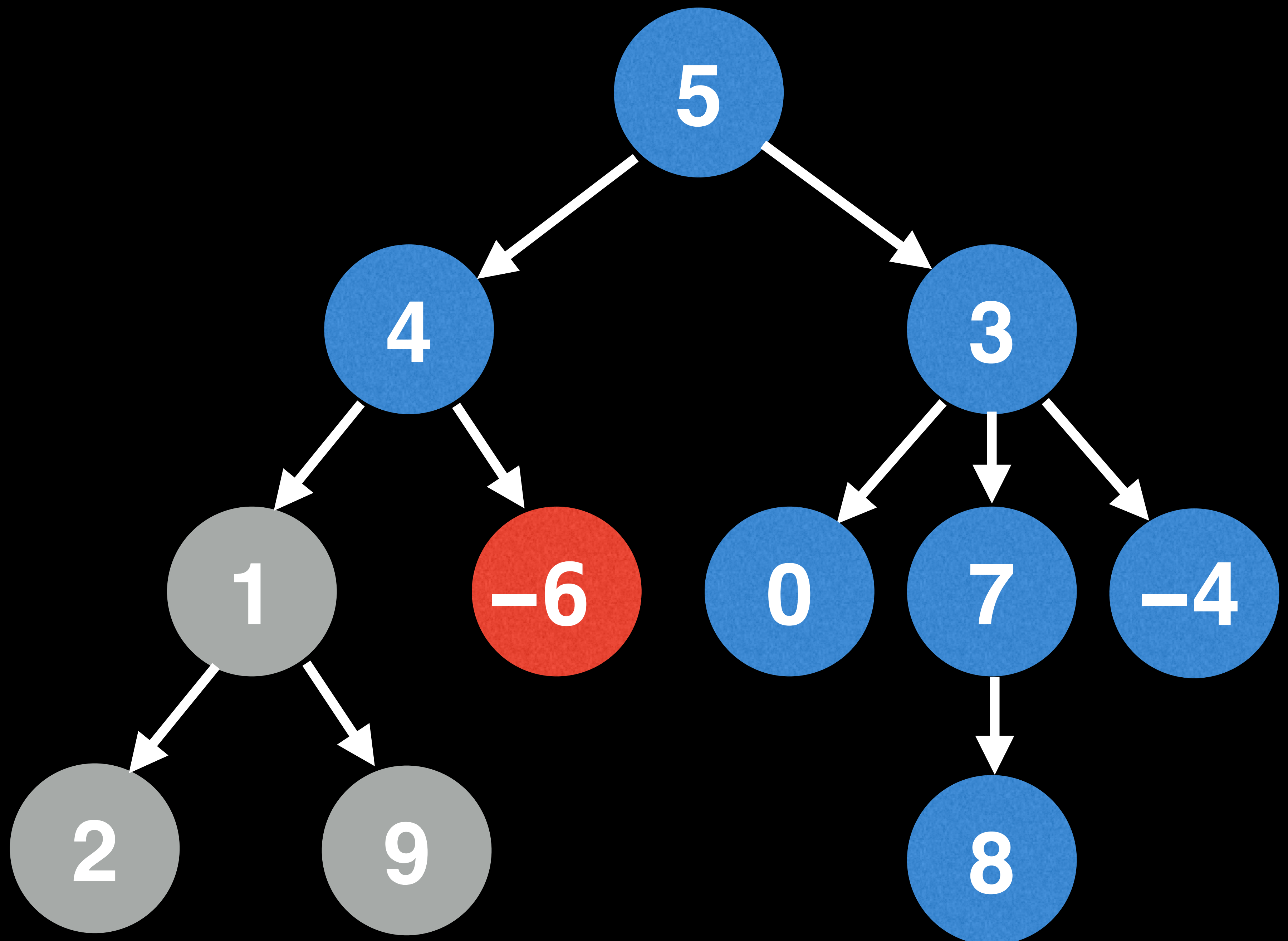
2



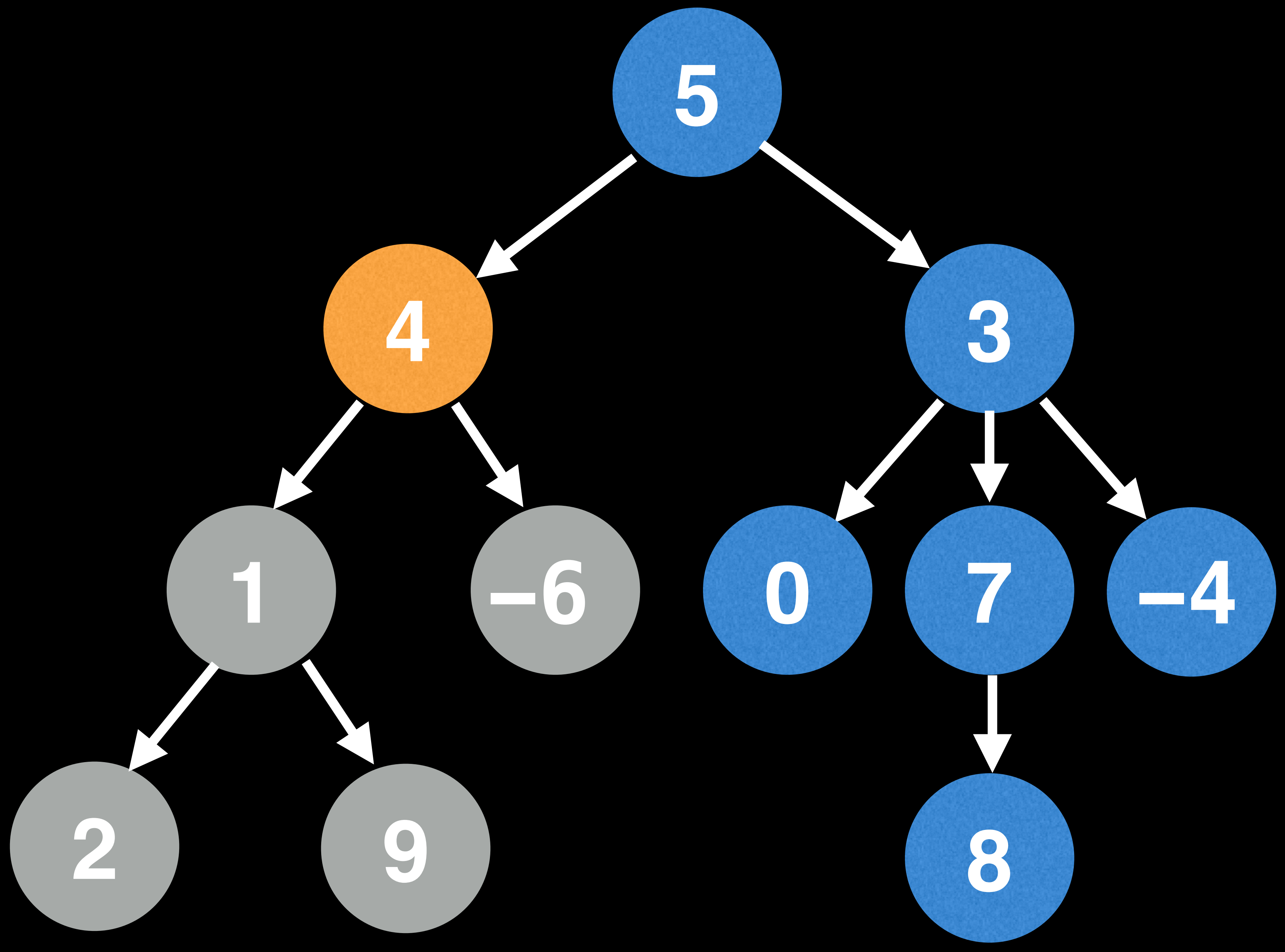
2 + 9



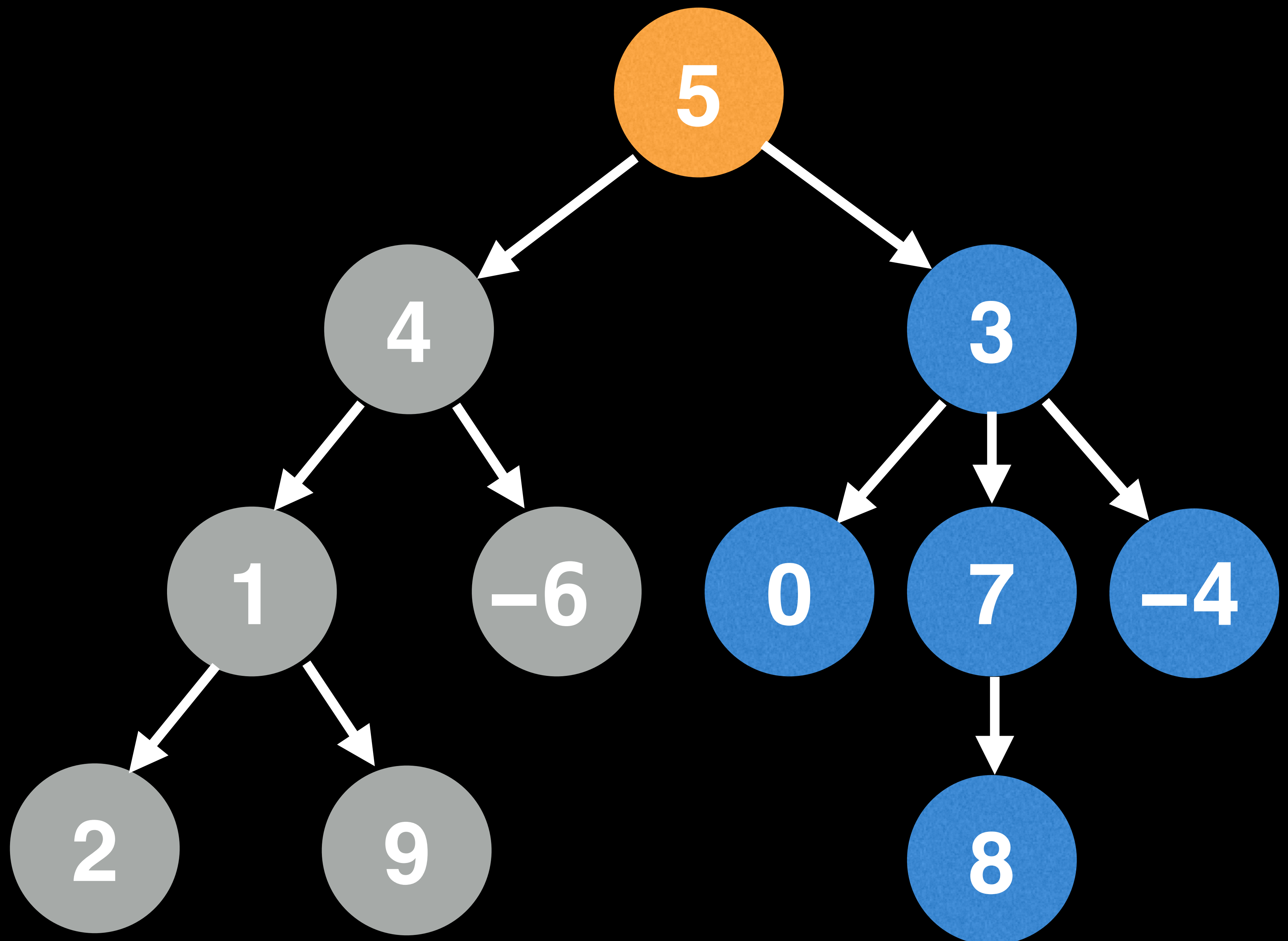
2 + 9



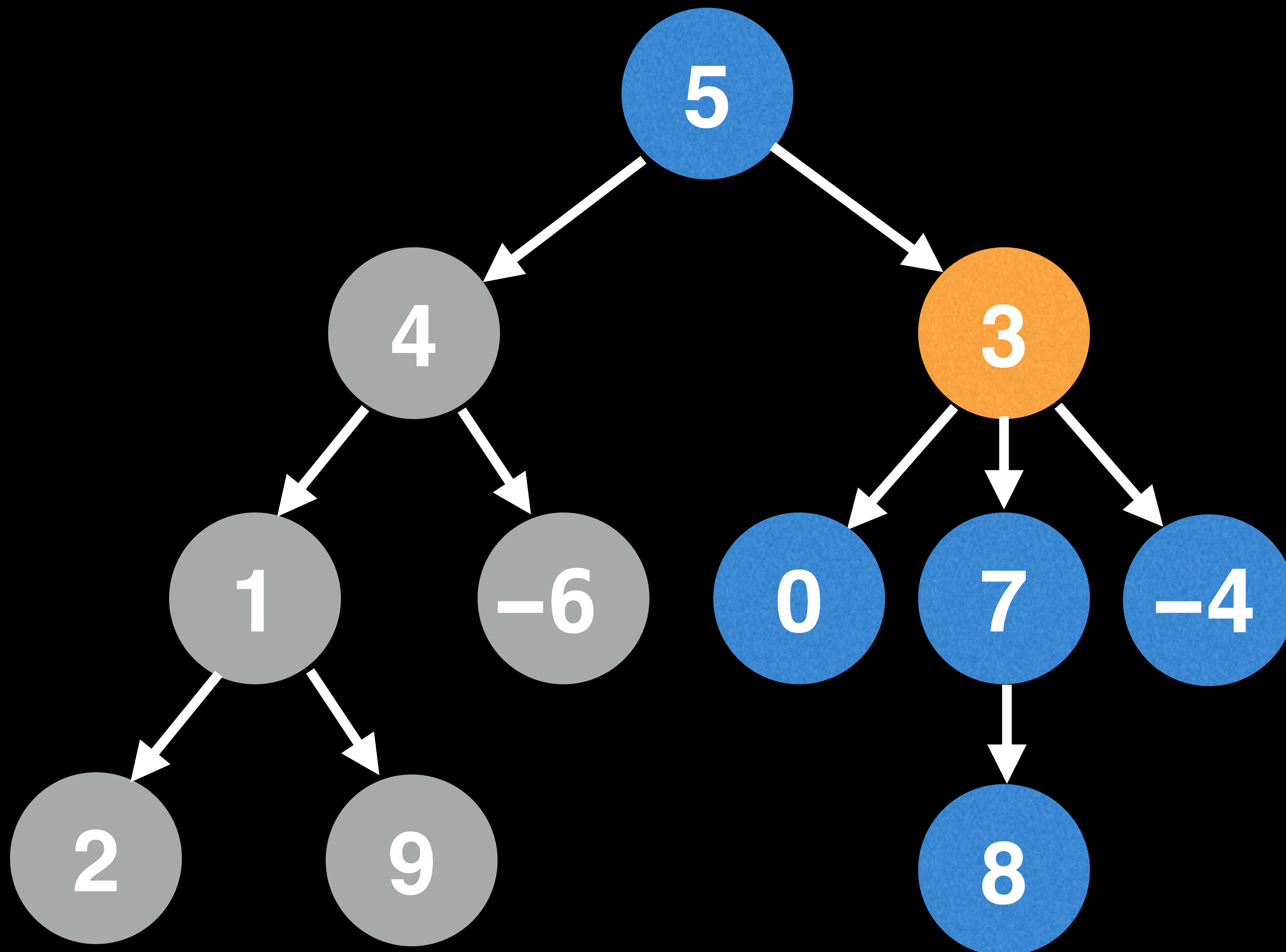
2 + 9



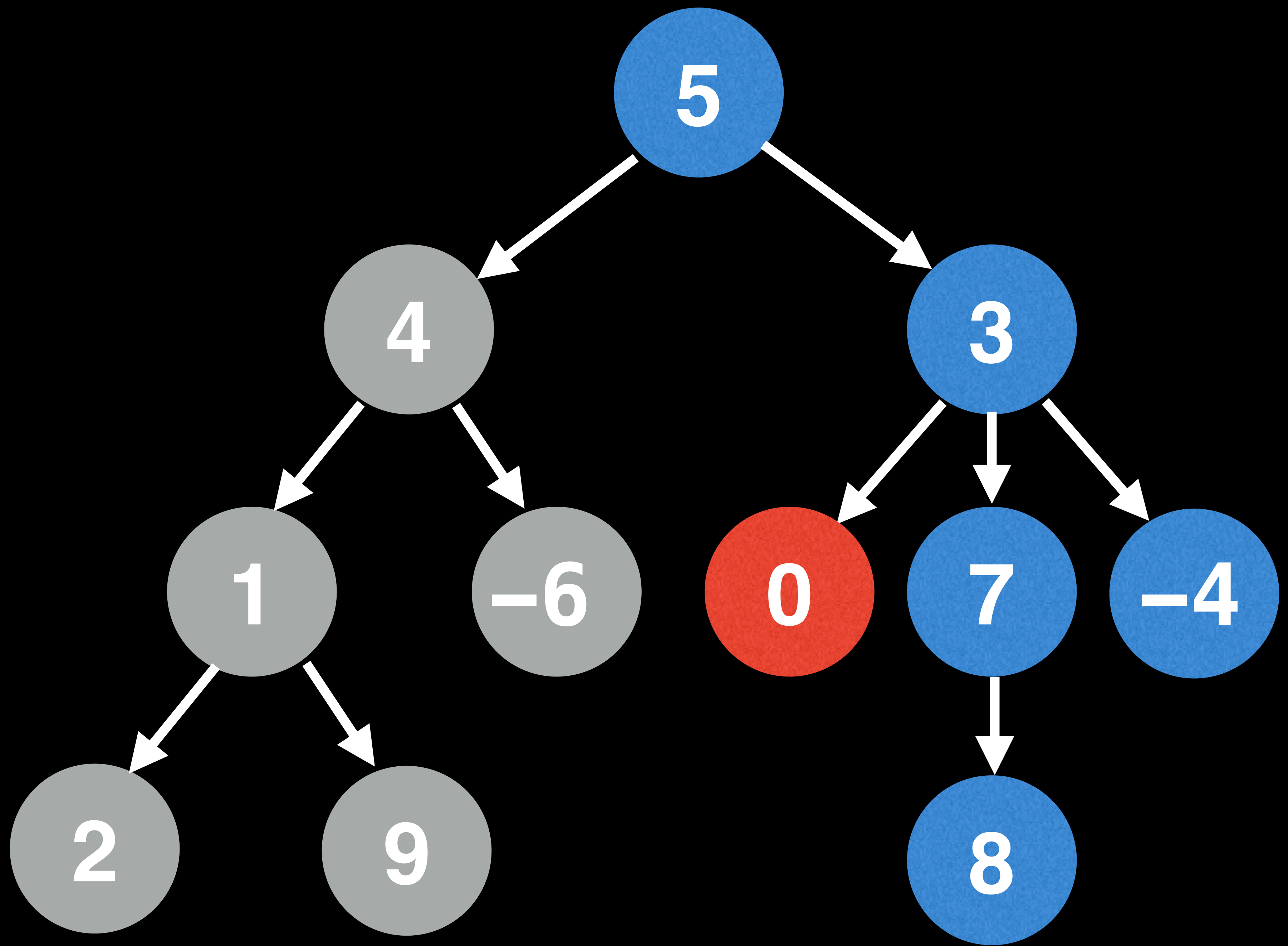
2 + 9 − 6



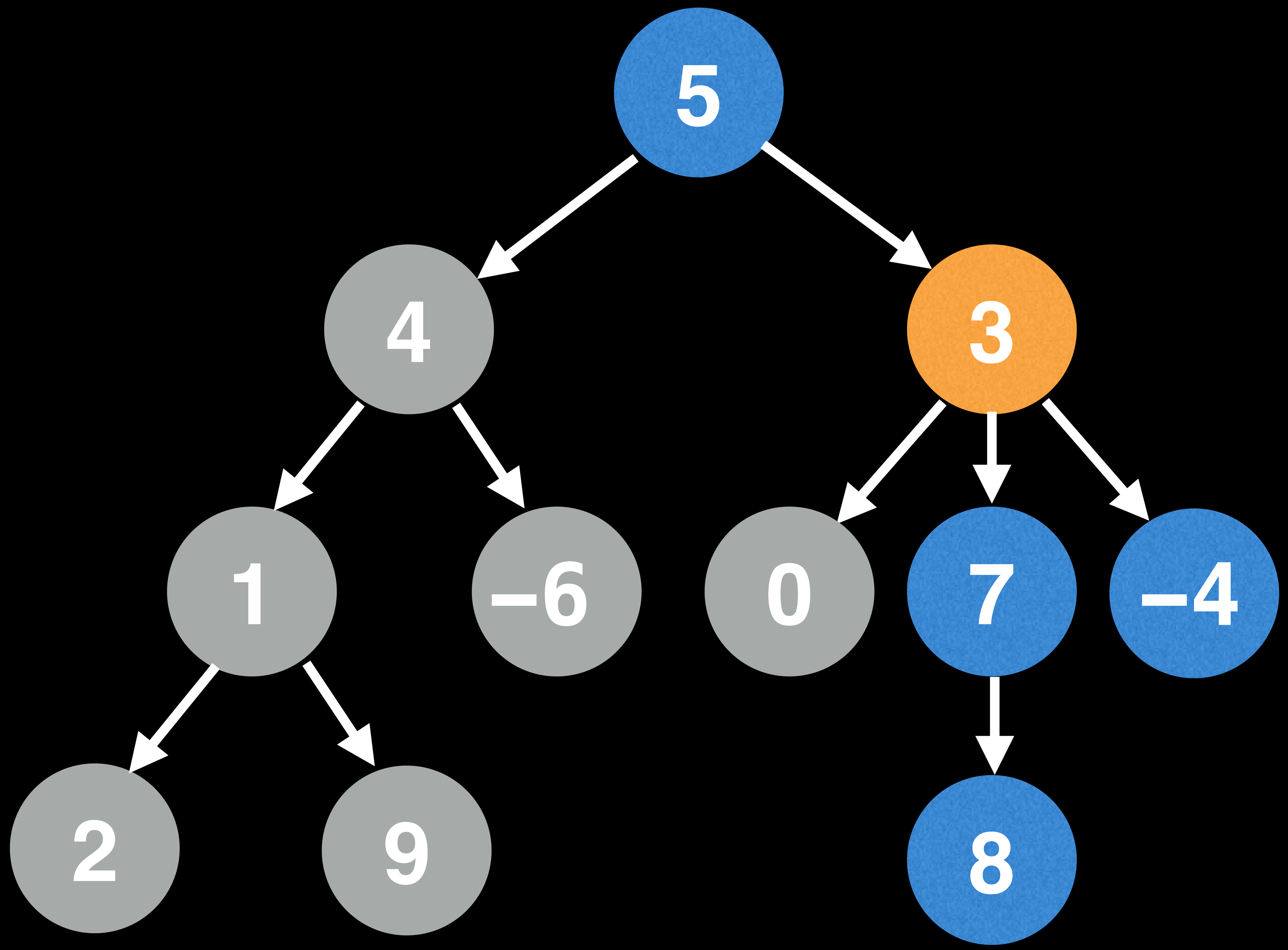
2 + 9 − 6



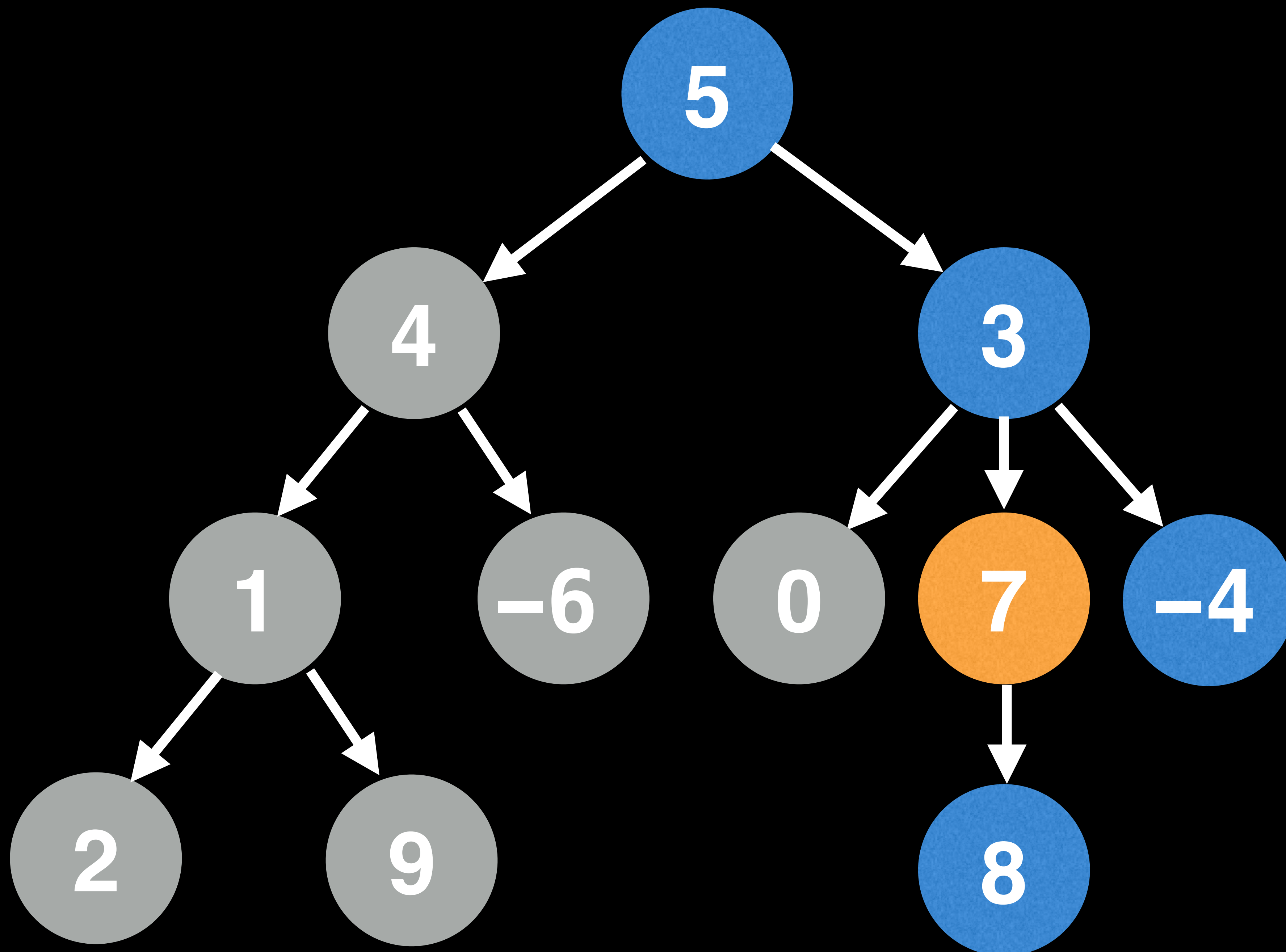
2 + 9 − 6



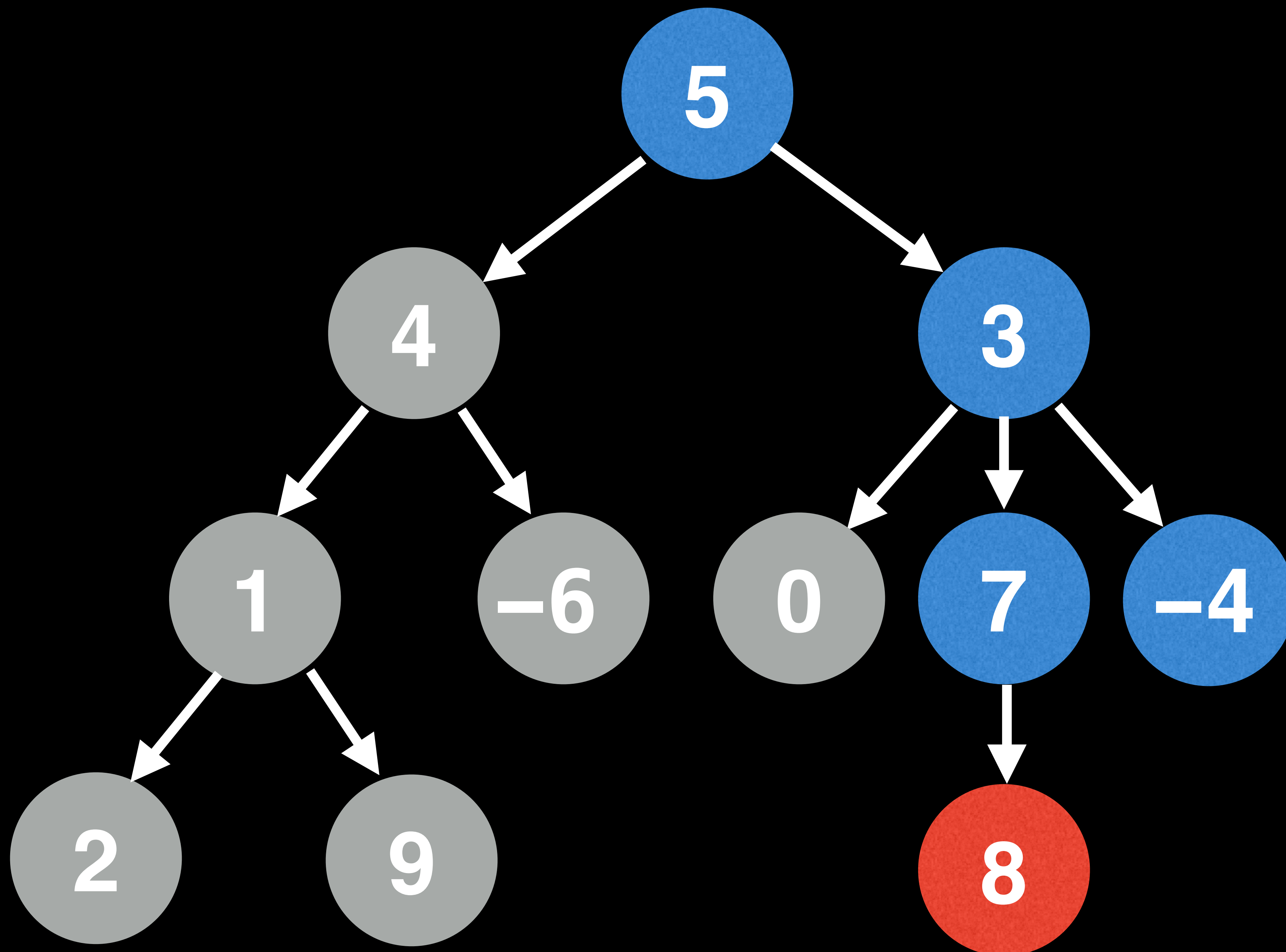
2 + 9 − 6



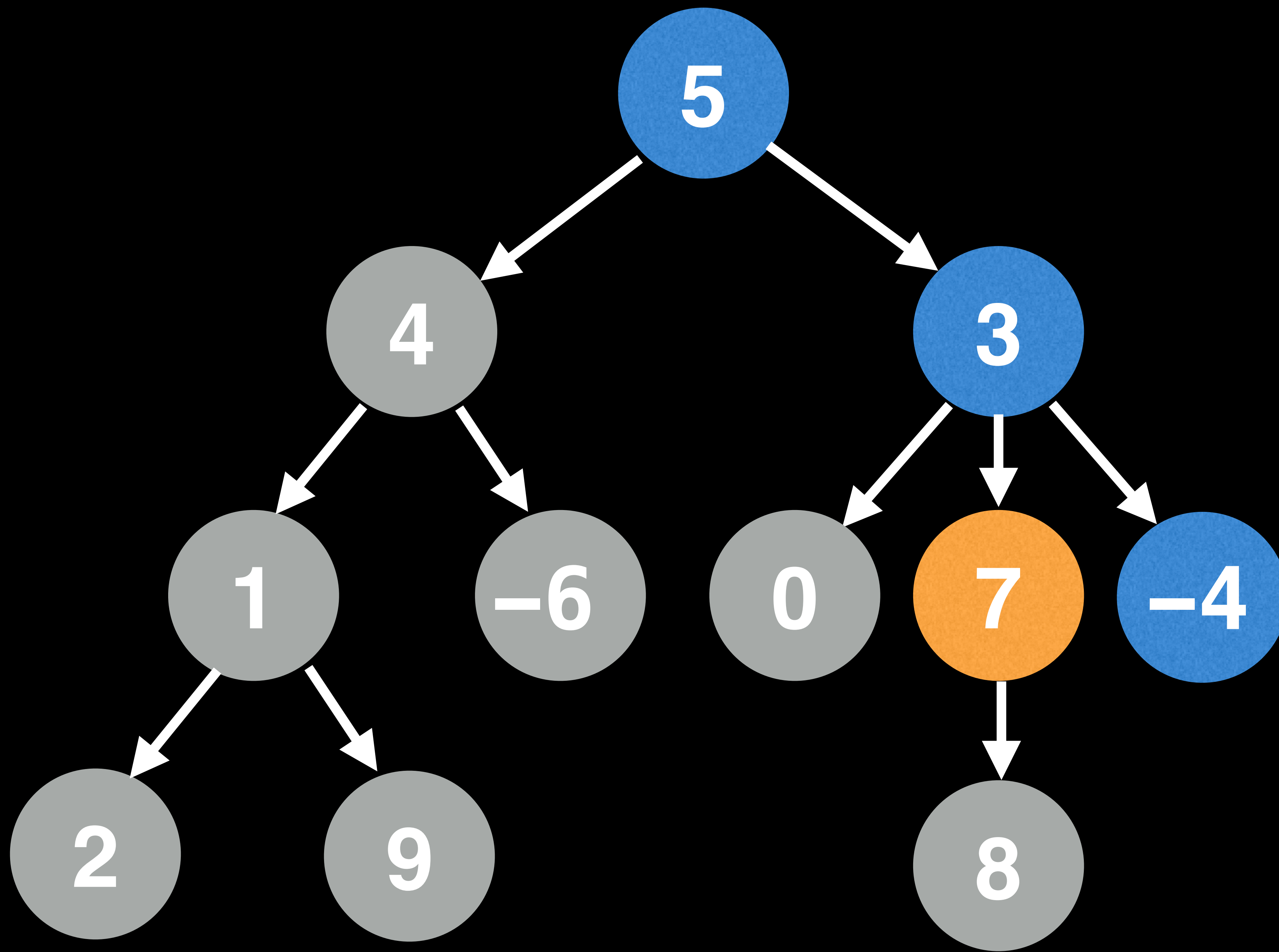
2 + 9 − 6 + 0



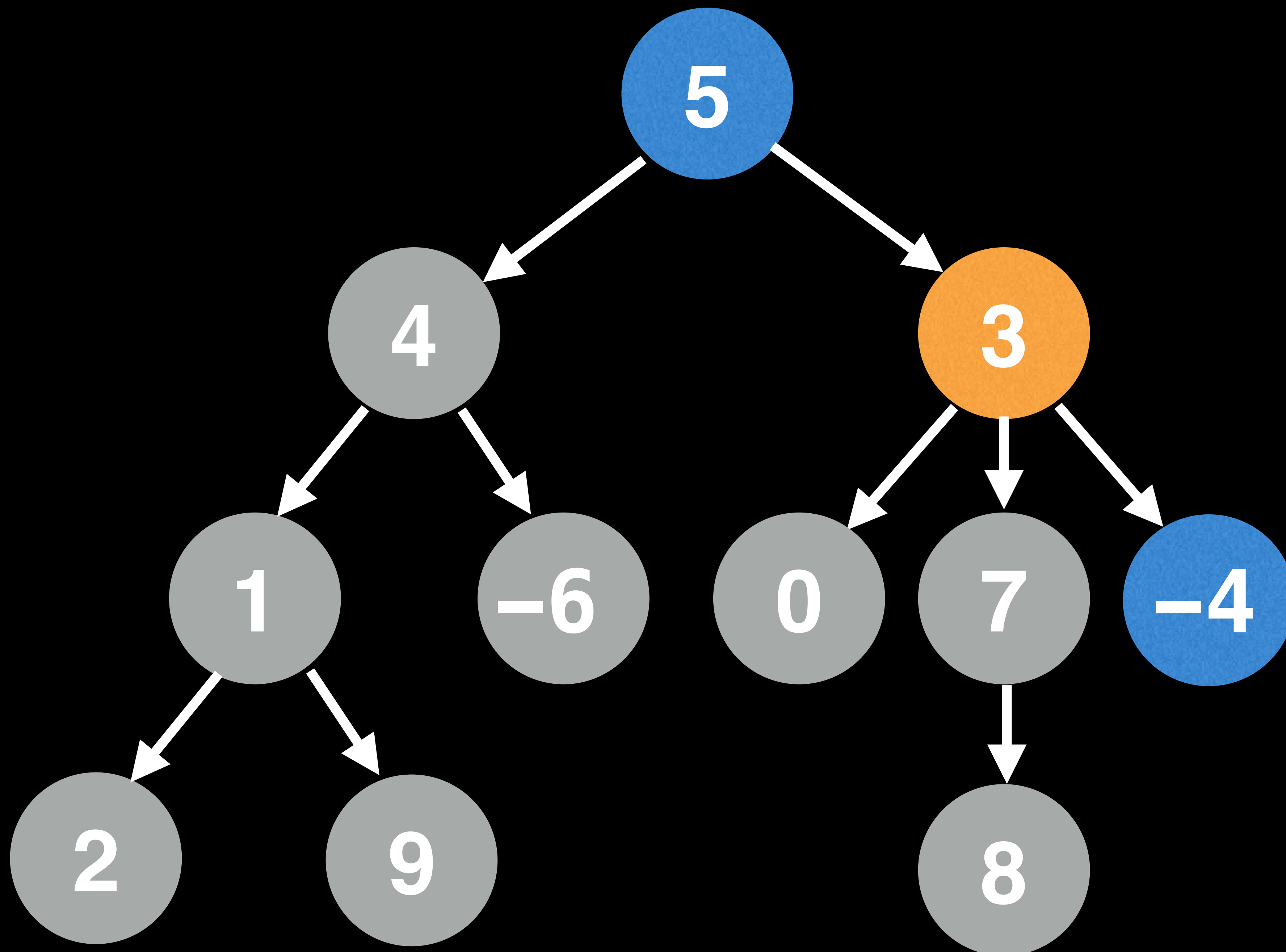
2 + 9 − 6 + 0



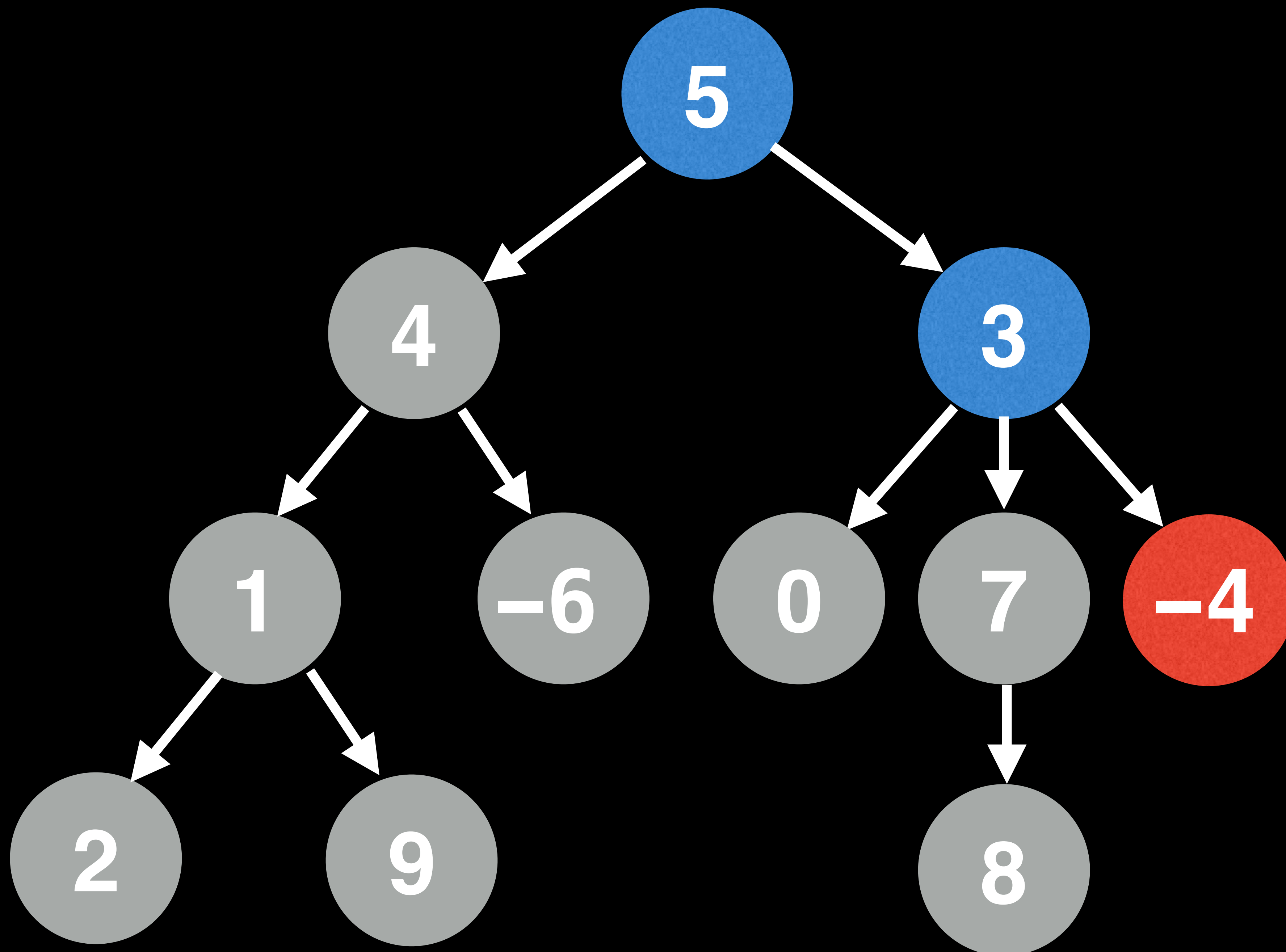
$2 + 9 - 6 + 0$



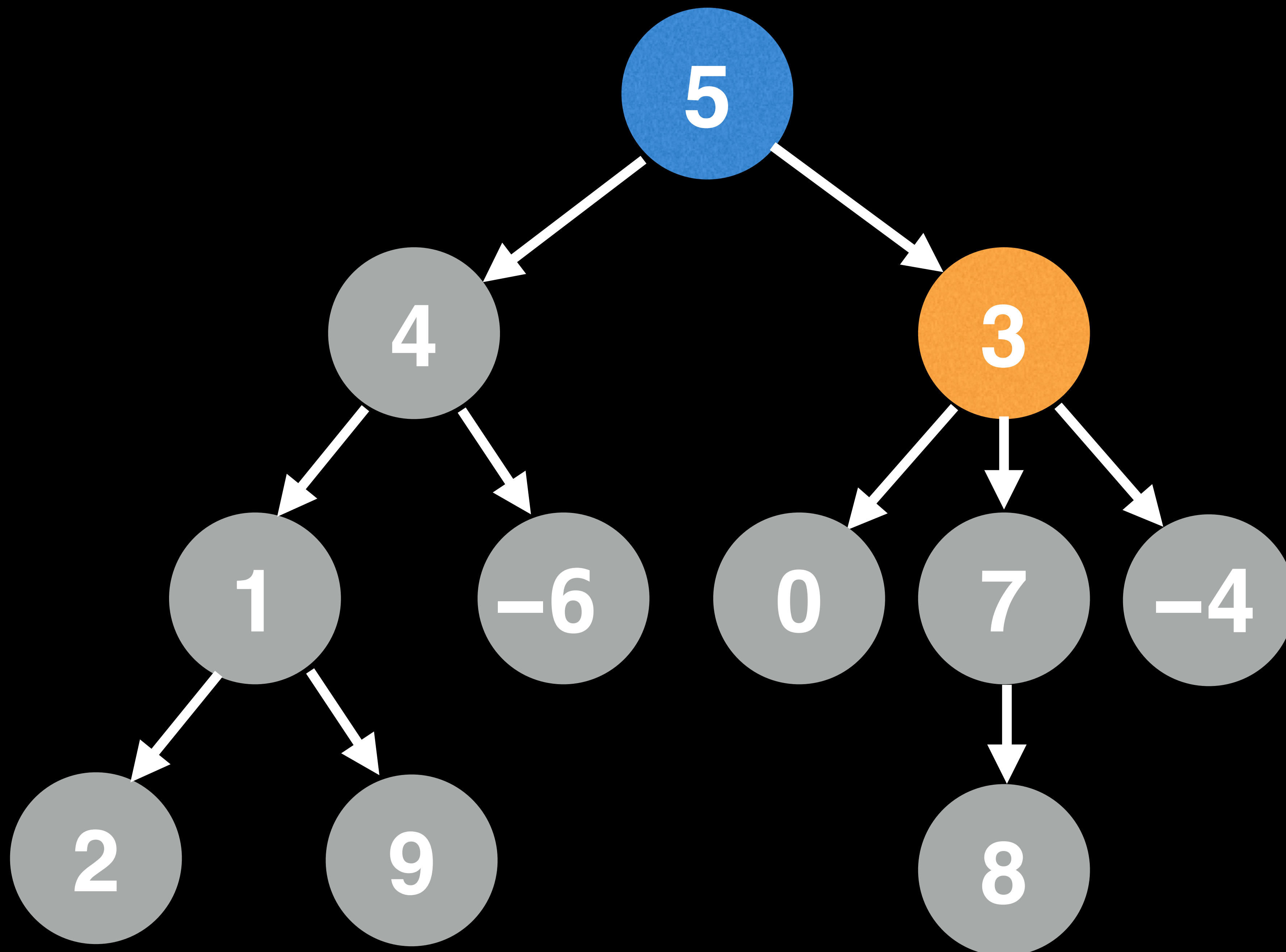
2 + 9 − 6 + 0 + 8



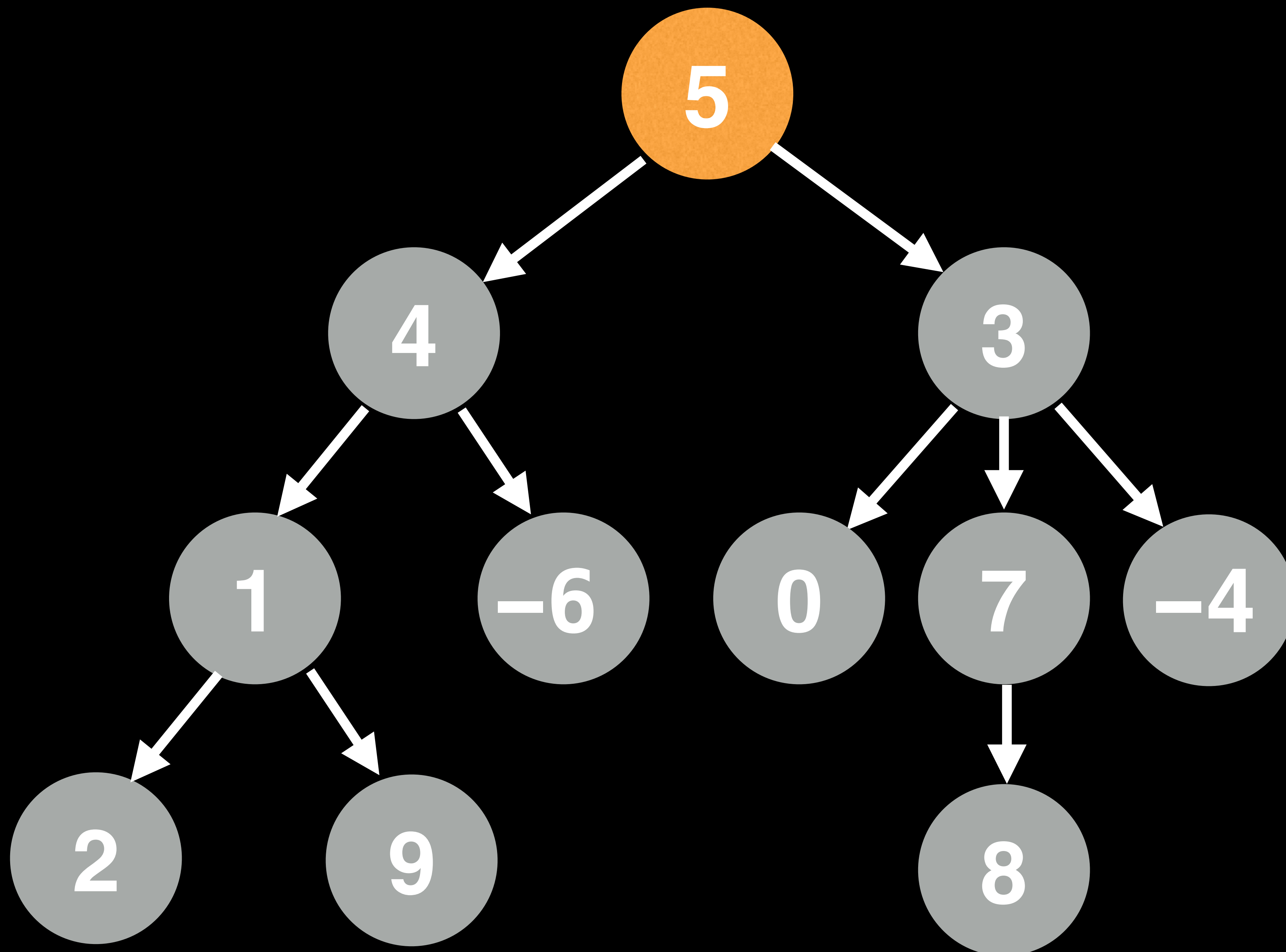
$2 + 9 - 6 + 0 + 8$



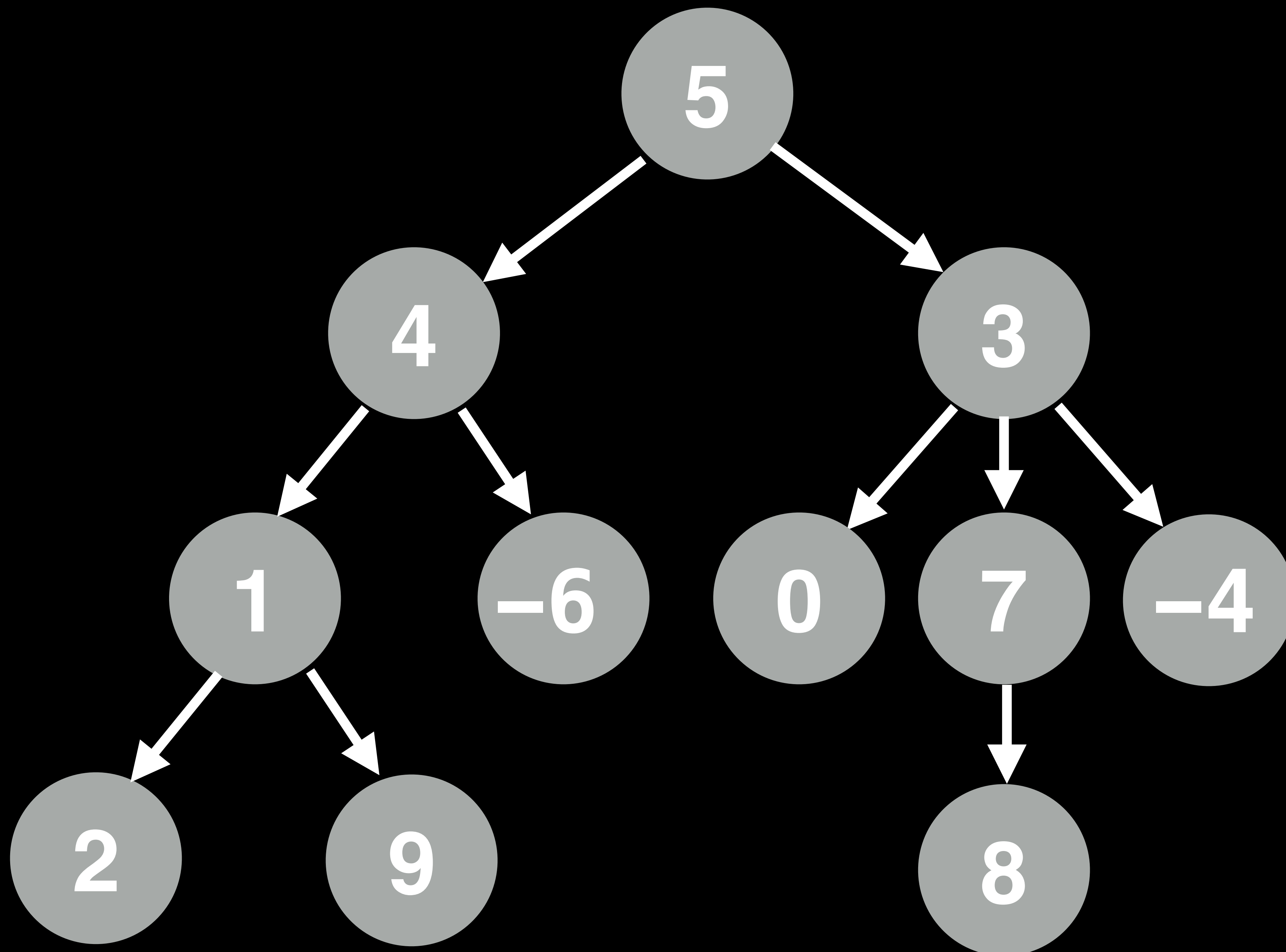
2 + 9 − 6 + 0 + 8



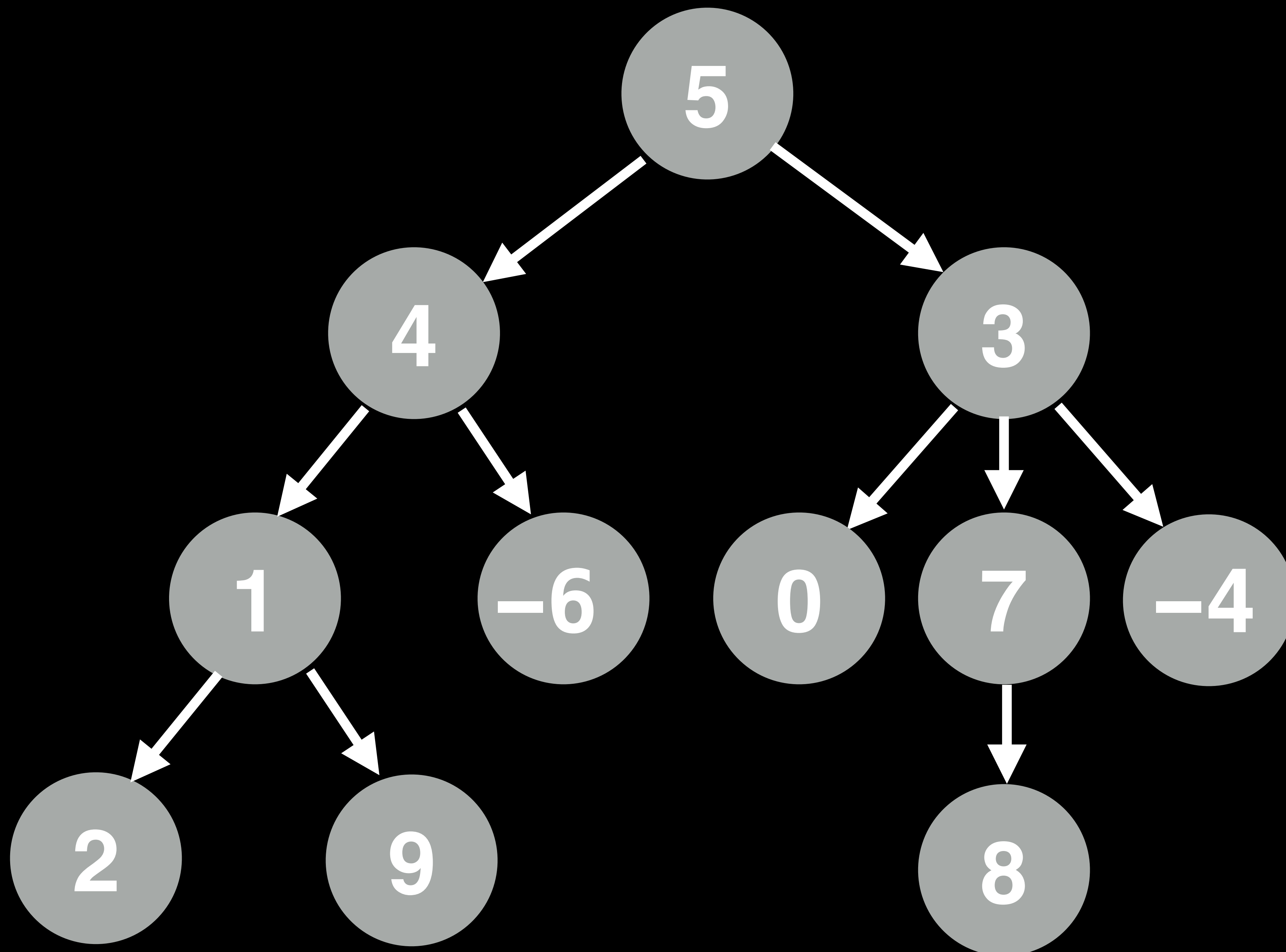
2 + 9 − 6 + 0 + 8 − 4



2 + 9 − 6 + 0 + 8 − 4



2 + 9 − 6 + 0 + 8 − 4



$$2 + 9 - 6 + 0 + 8 - 4 = 9$$

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

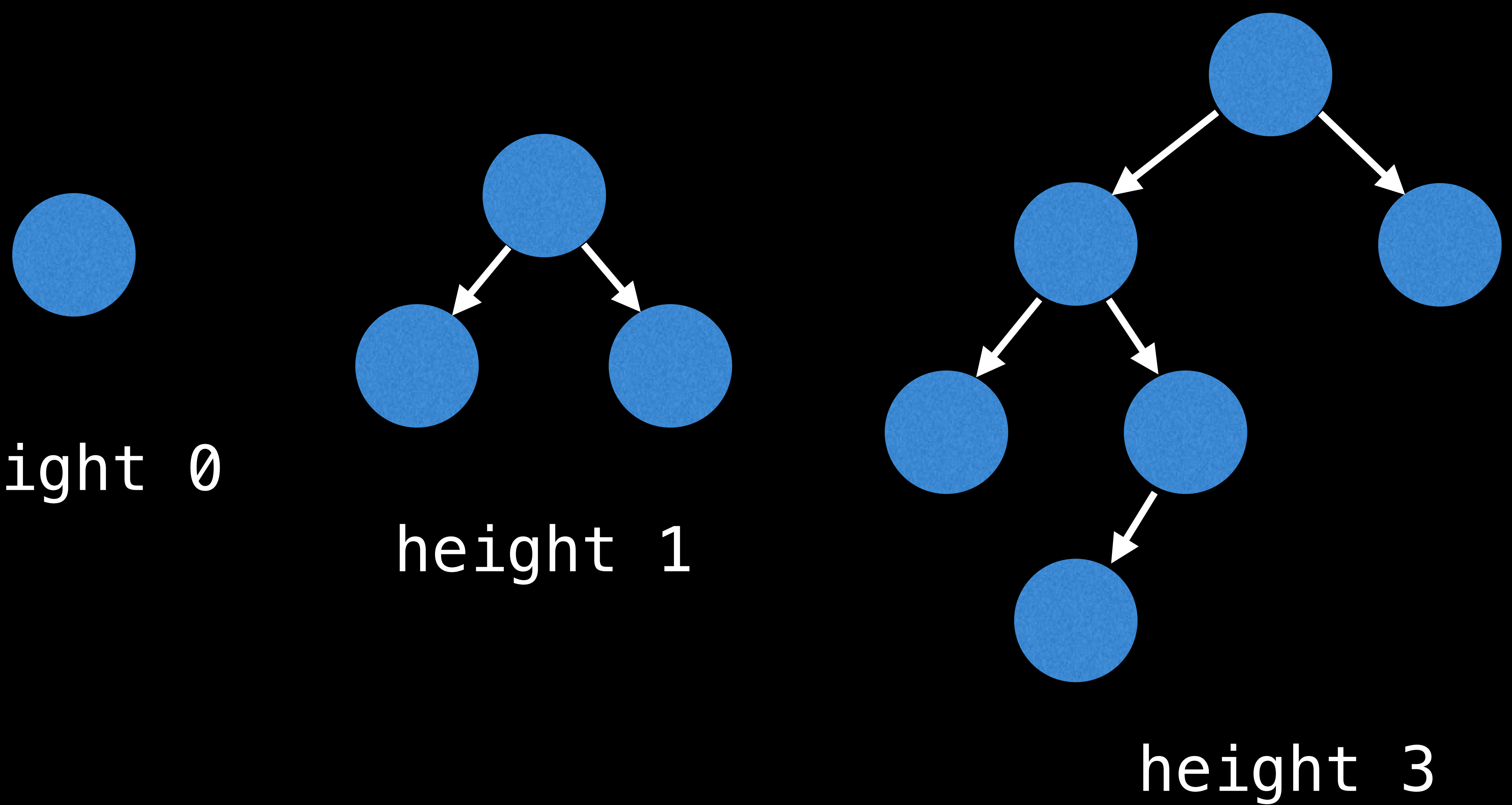
```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

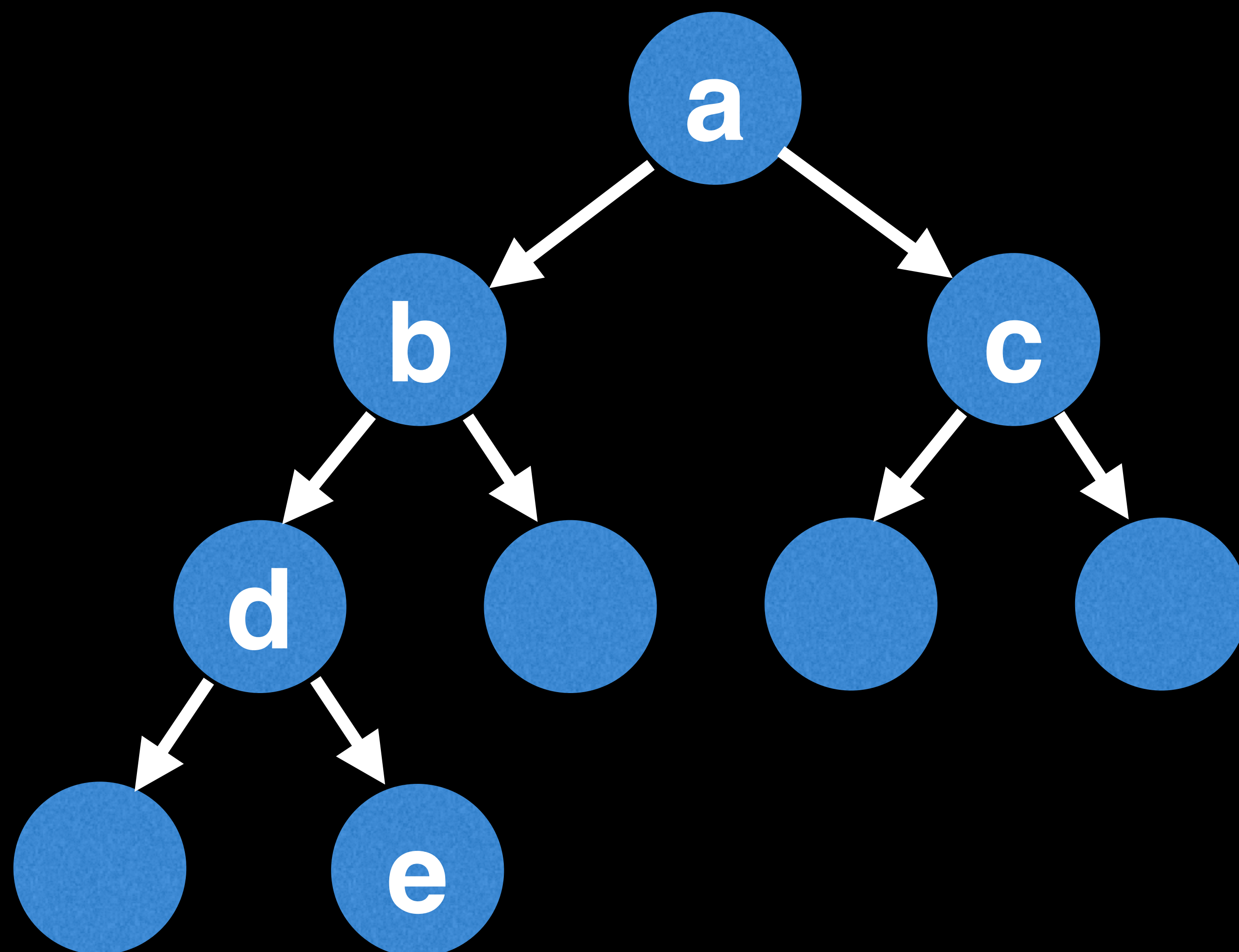
```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

Problem 2: Tree Height

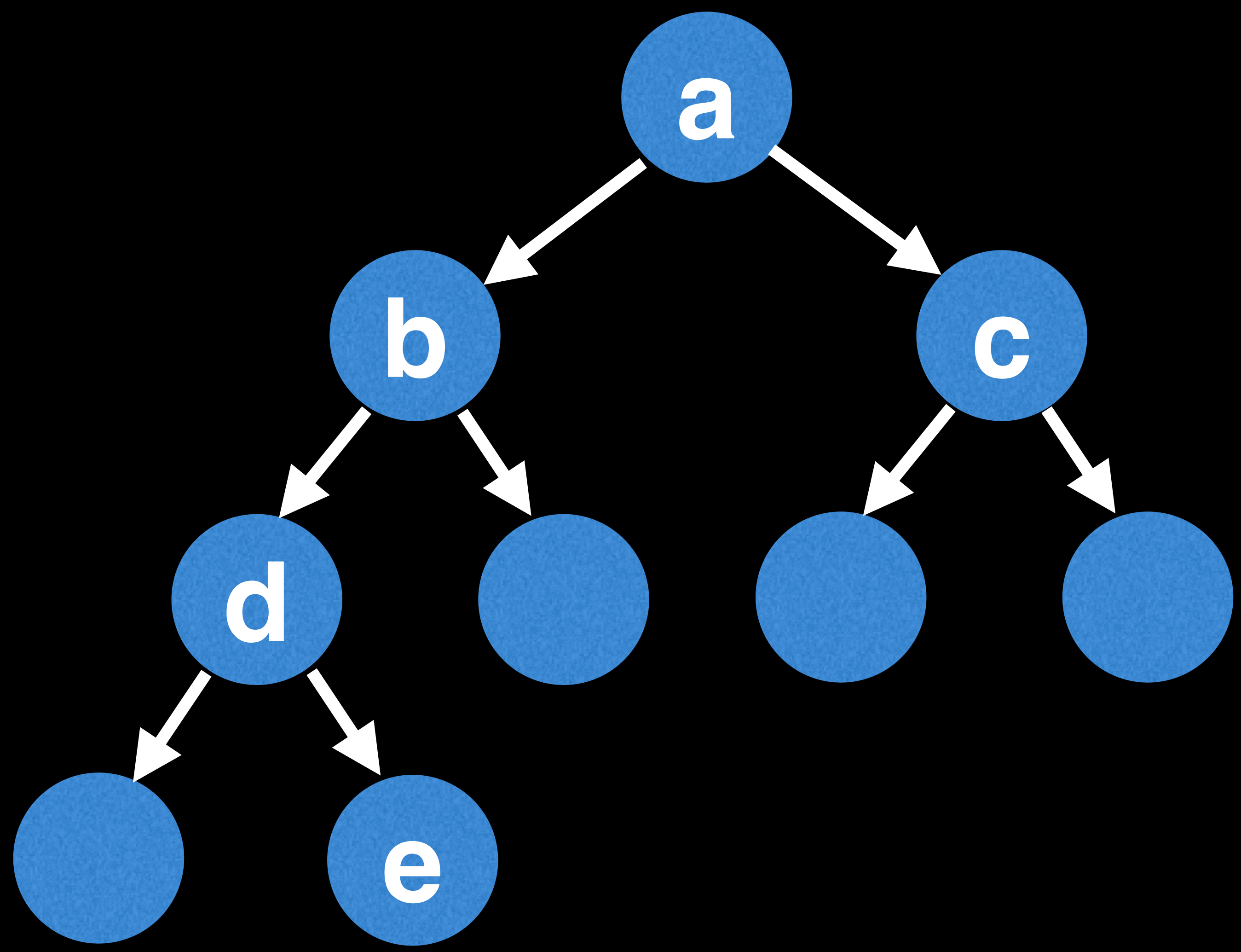
Find the **height** of a **binary tree**. The **height** of a tree is the number of edges from the root to the lowest leaf.



Let **h**(x) be the height of the subtree rooted at node x.

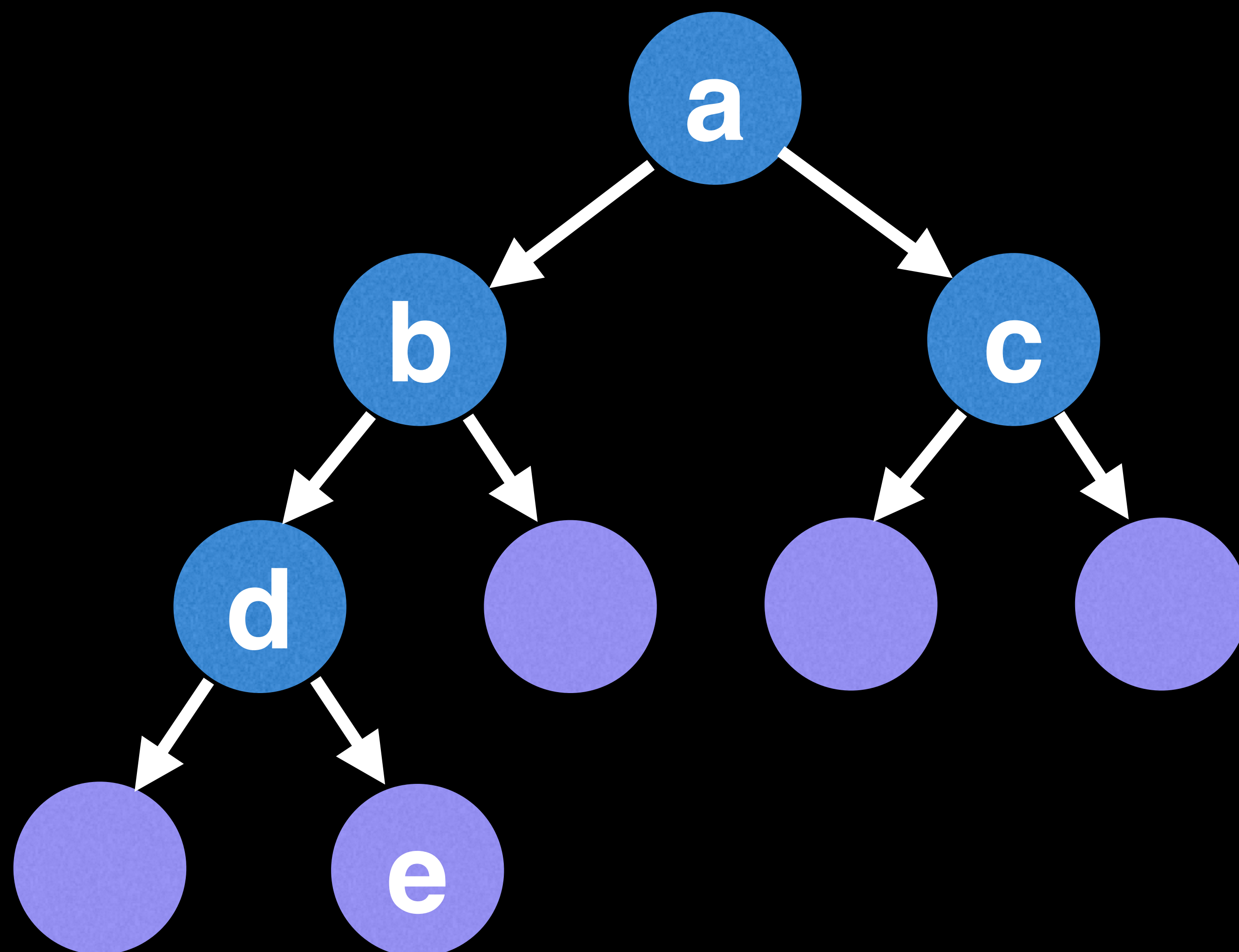


Let **h**(x) be the height of the subtree rooted at node x.



h(a) = 3, **h**(b) = 2, **h**(c) = 1, **h**(d) = 1, **h**(e) = 0

By themselves, leaf nodes such as node **e** don't have children, so they don’t add any additional height to the tree.

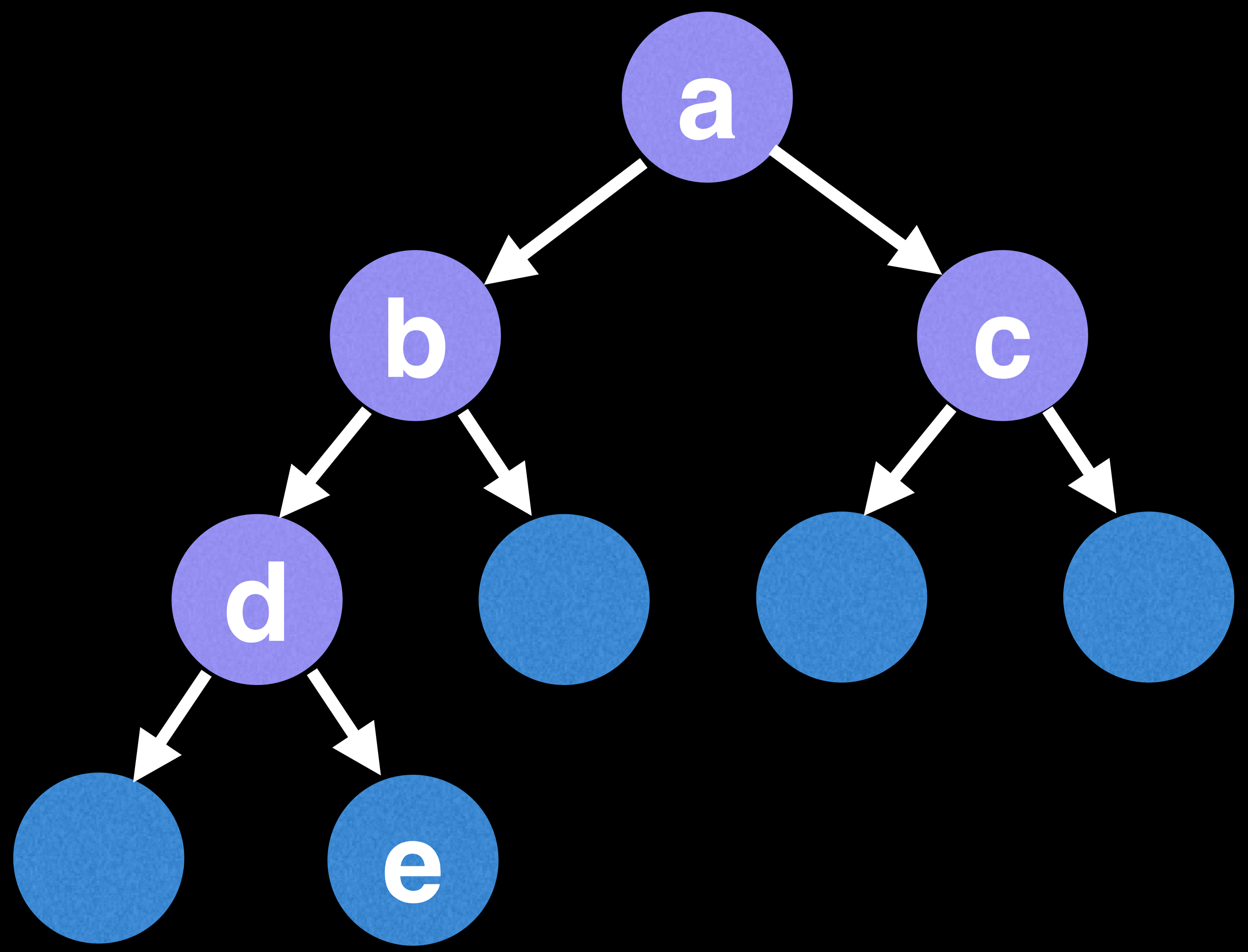


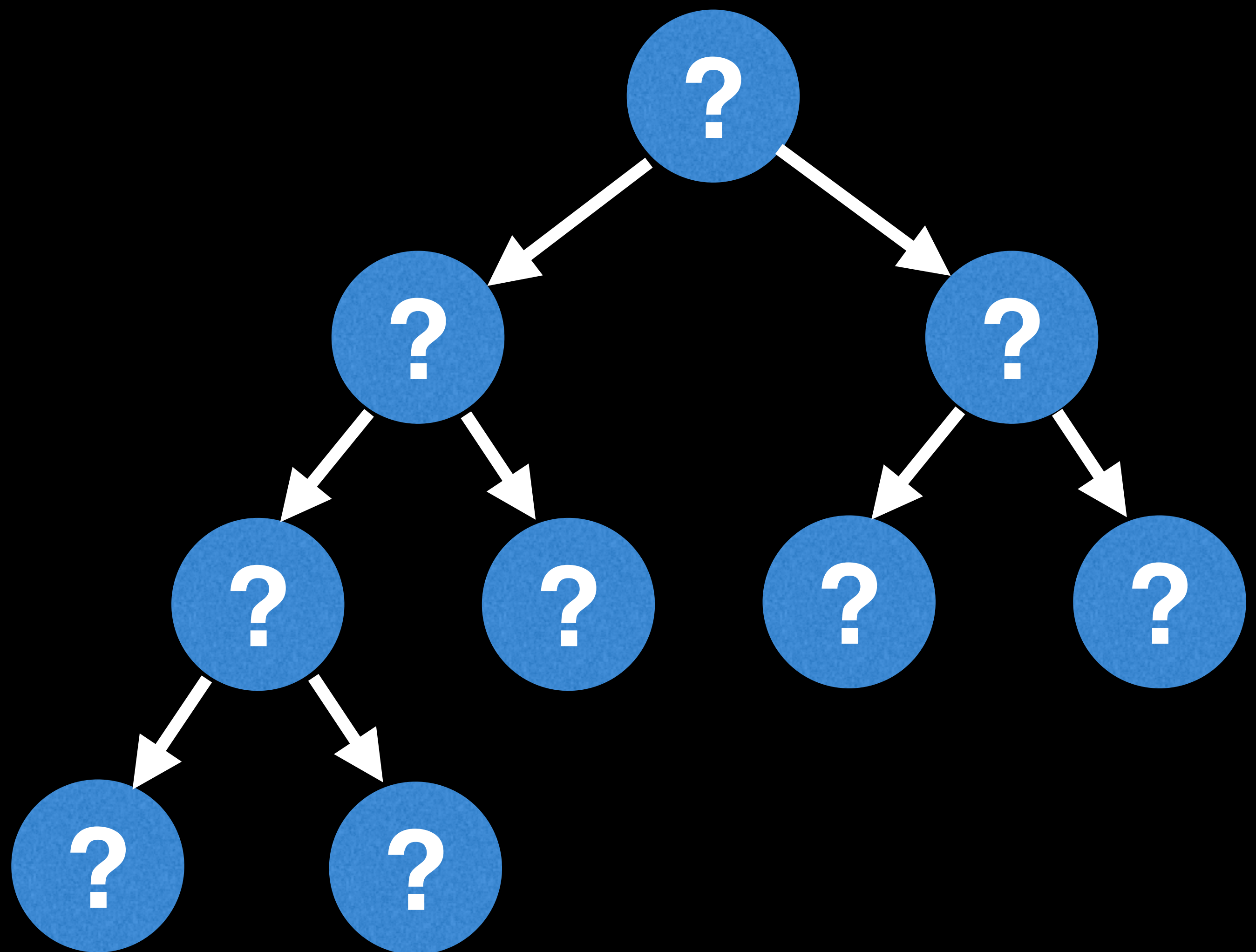
As a base case we can conclude that:

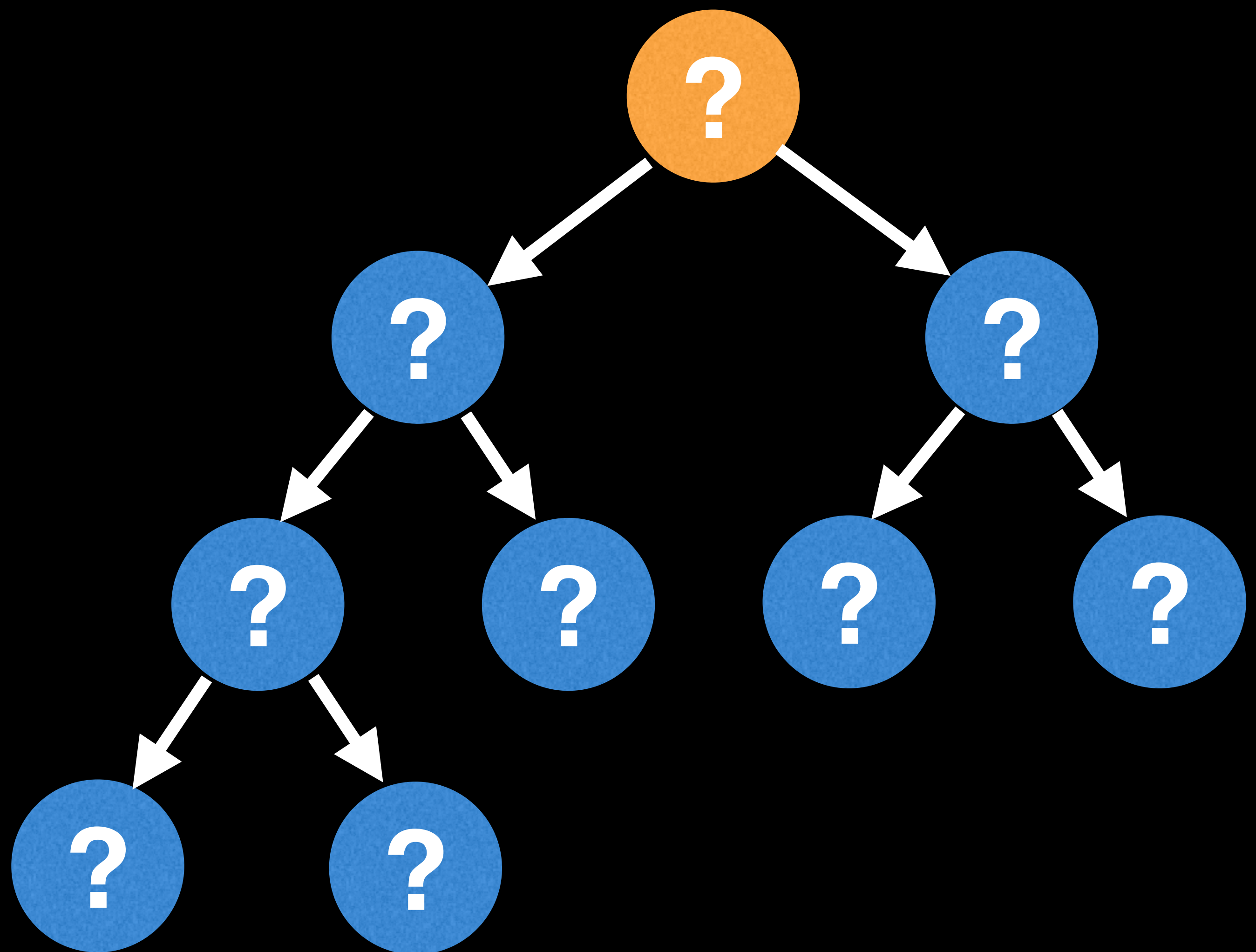
h(leaf node) = 0

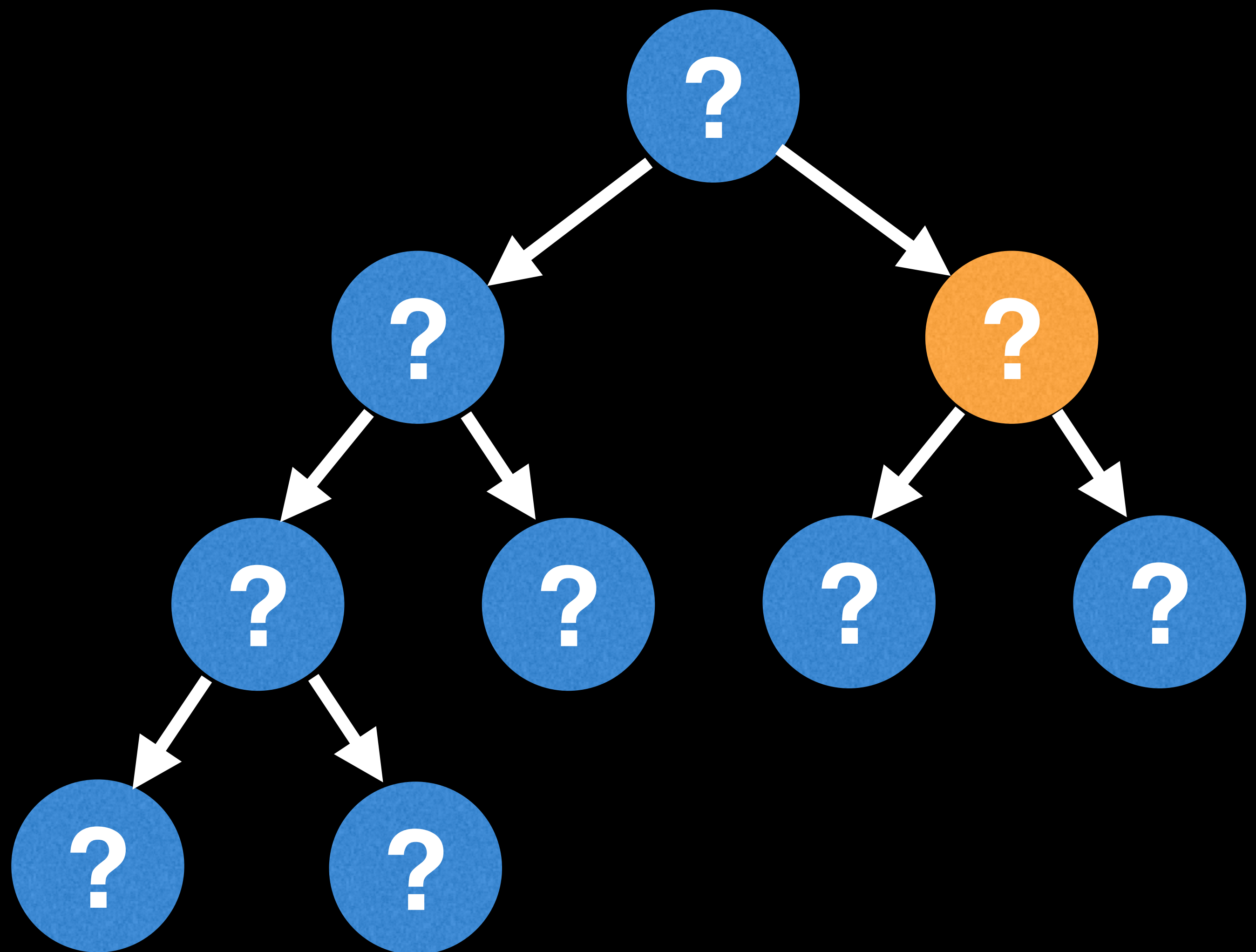
Assuming node x is not a leaf node, we're able to formulate a recurrence for the height:

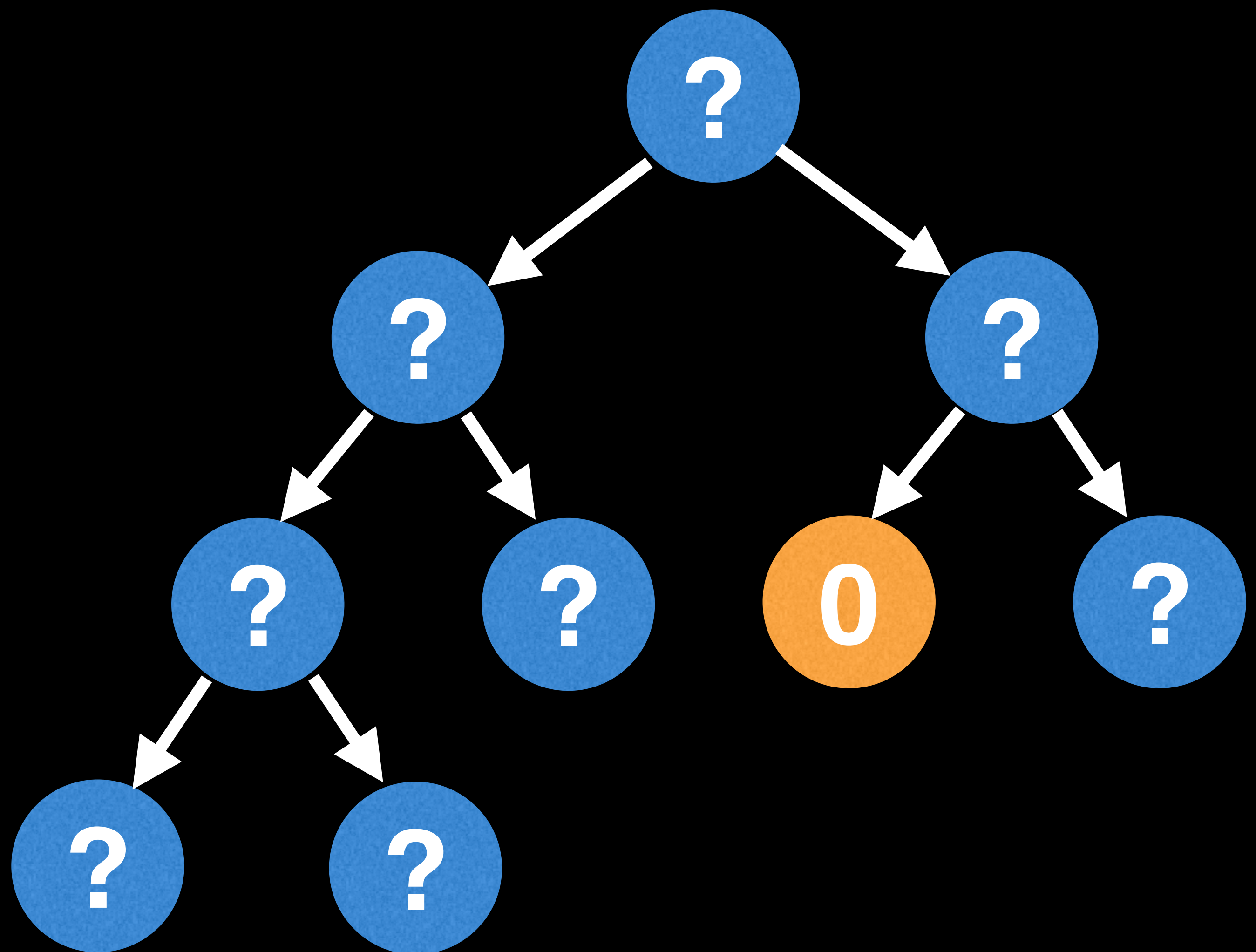
$$h(x) = \max(h(x.left), h(x.right)) + 1$$



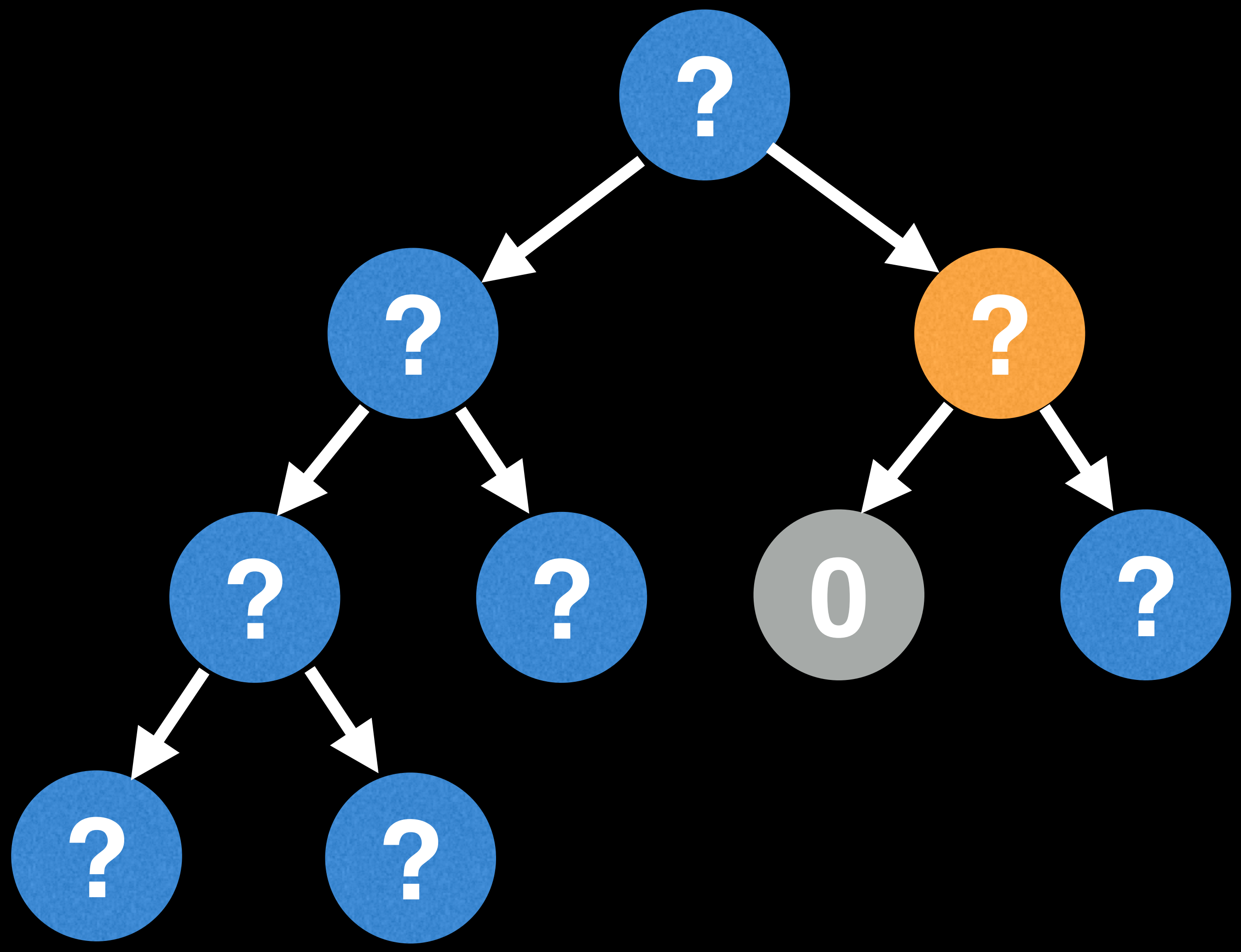


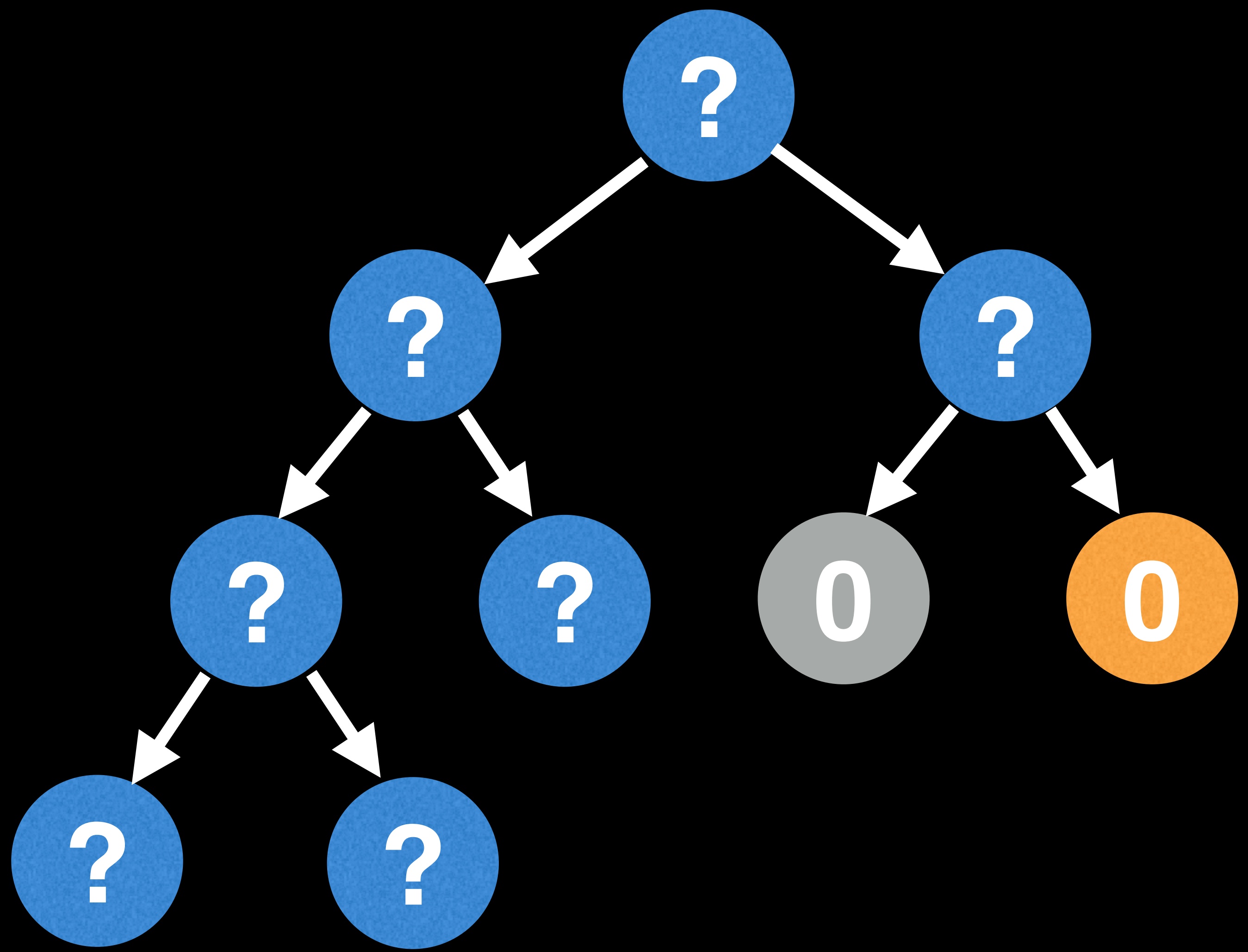




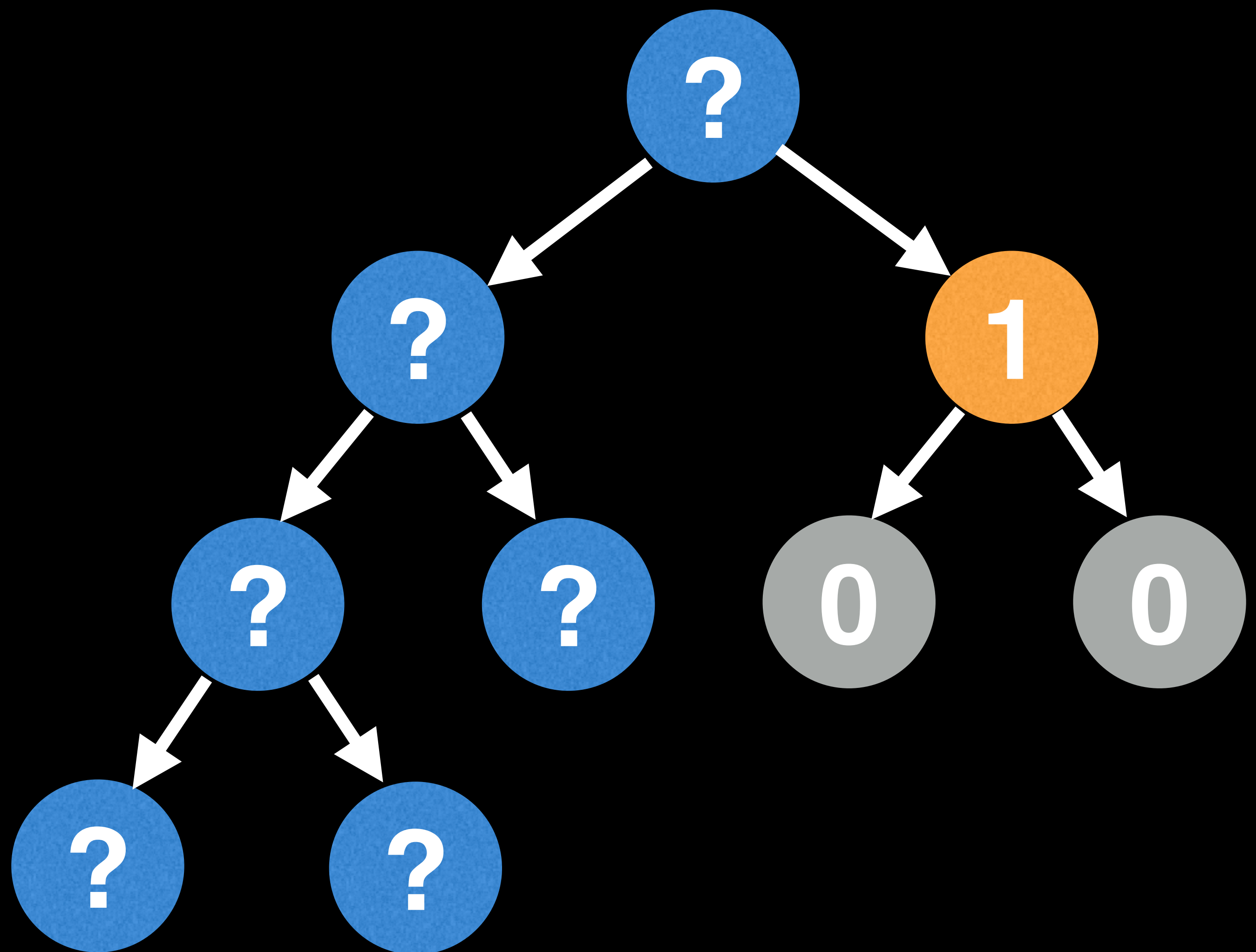


Leaf node has a height of 0

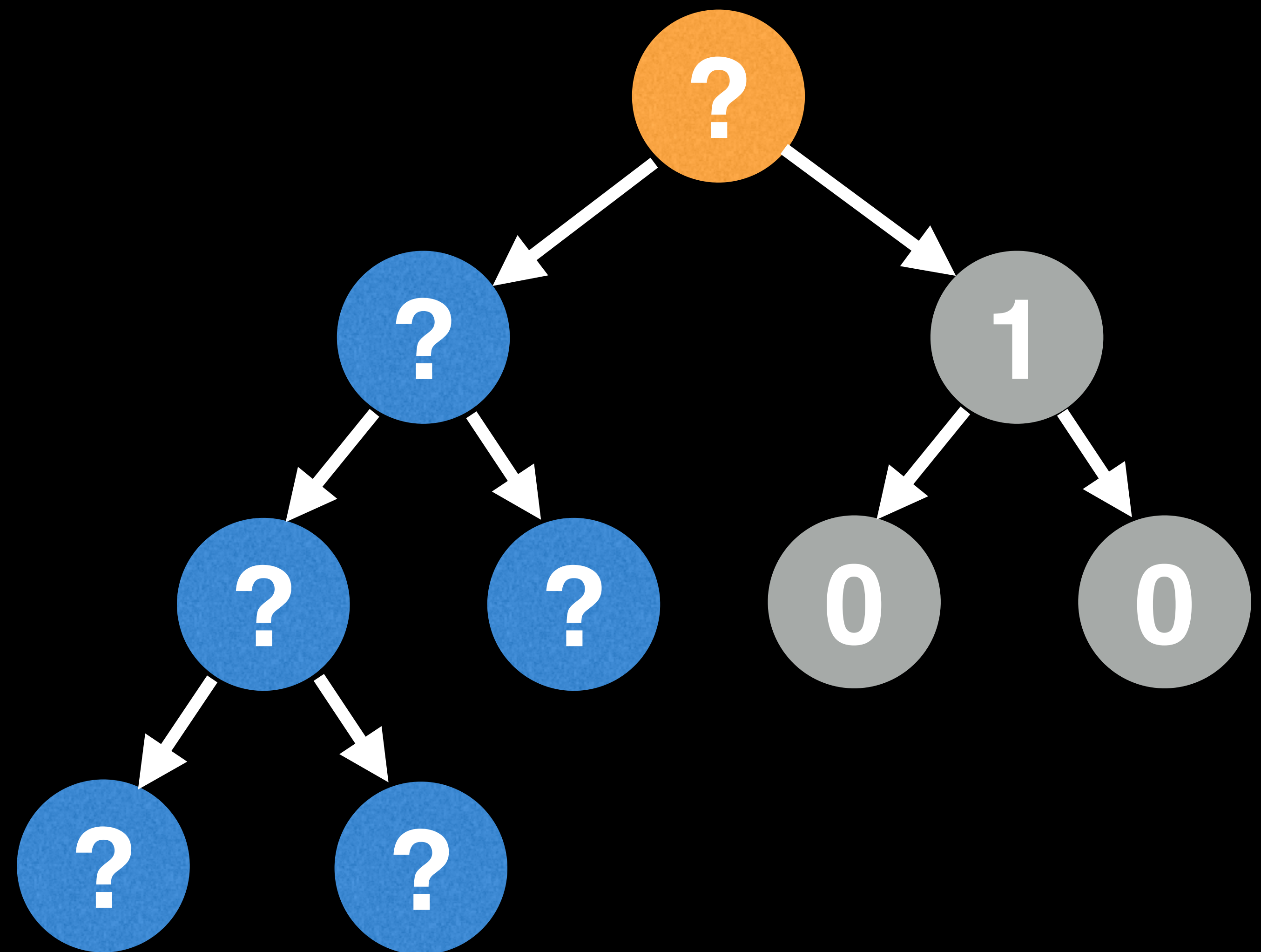

?
?
?
?
?
0
?
?
?

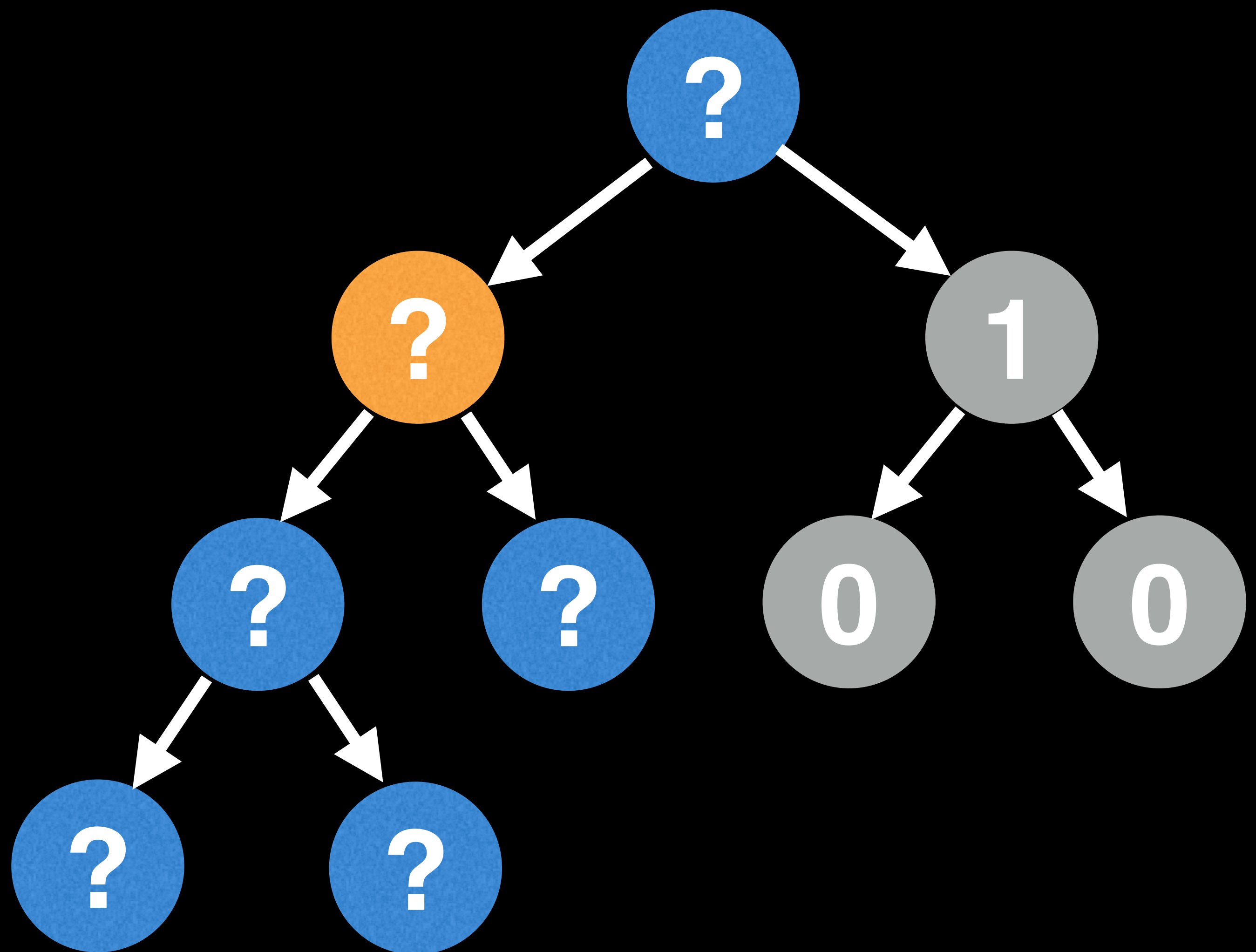


Leaf node has a height of 0

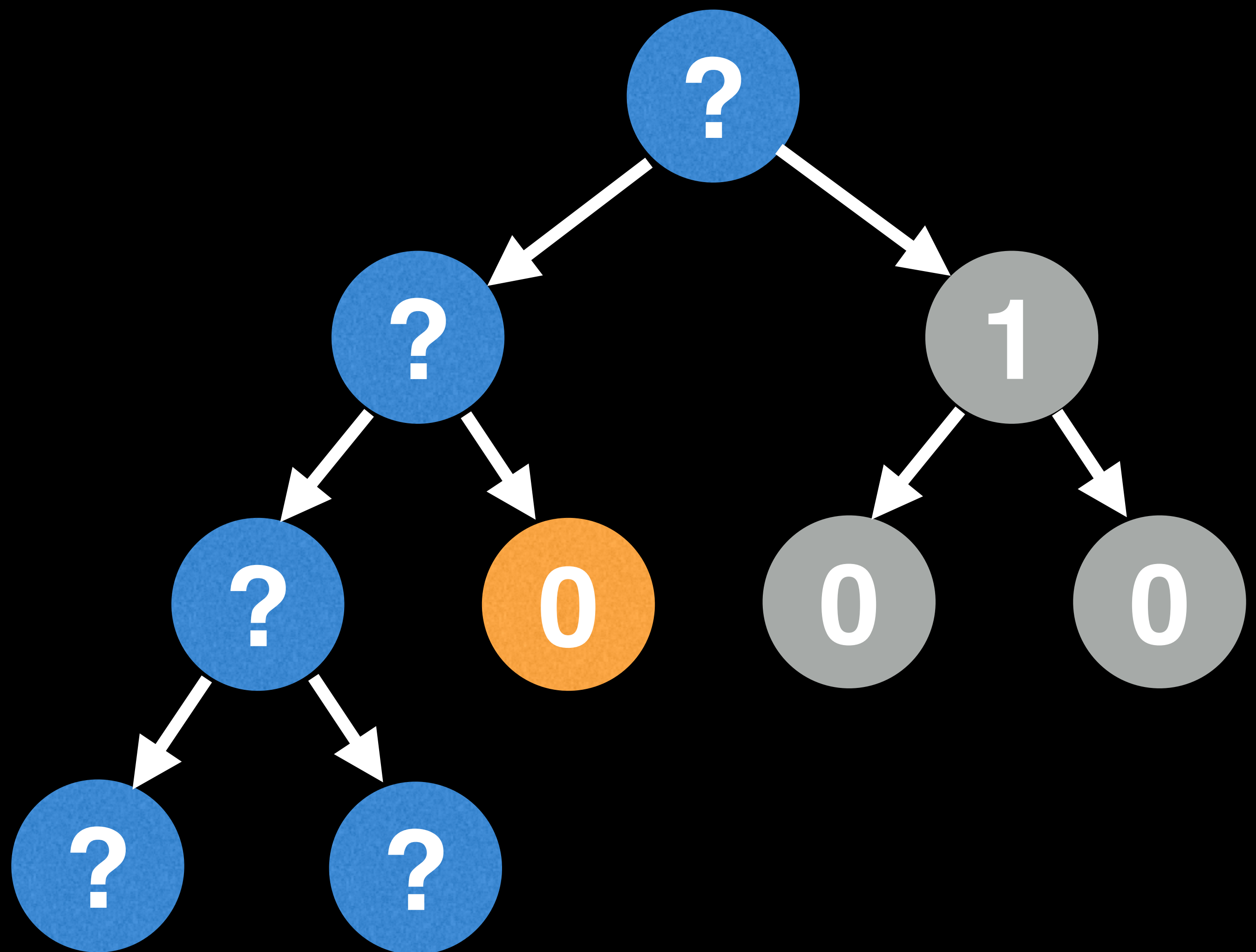


height = **max**(0, 0) + 1 = 1

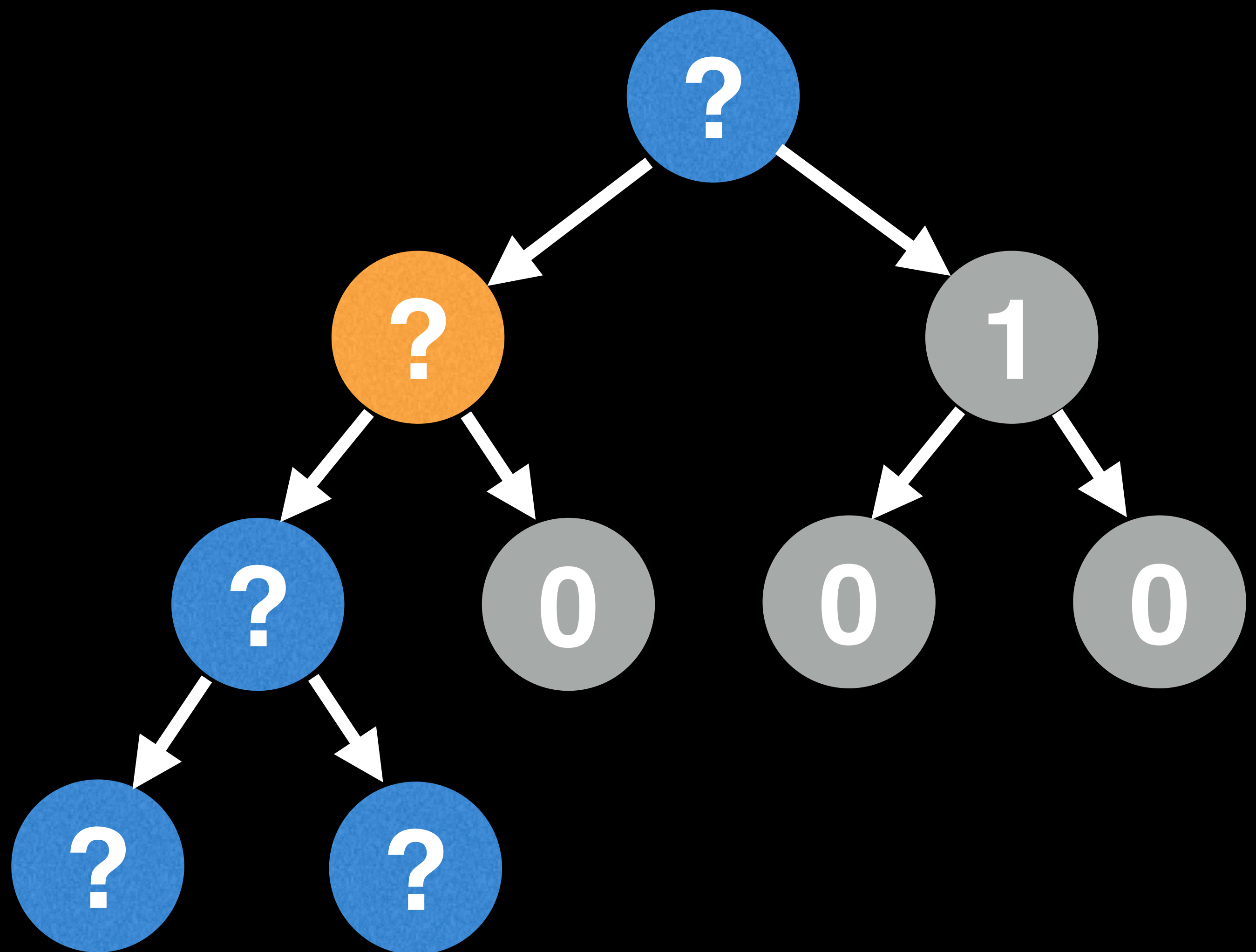

?
?
1
?
?
0
0
?
?

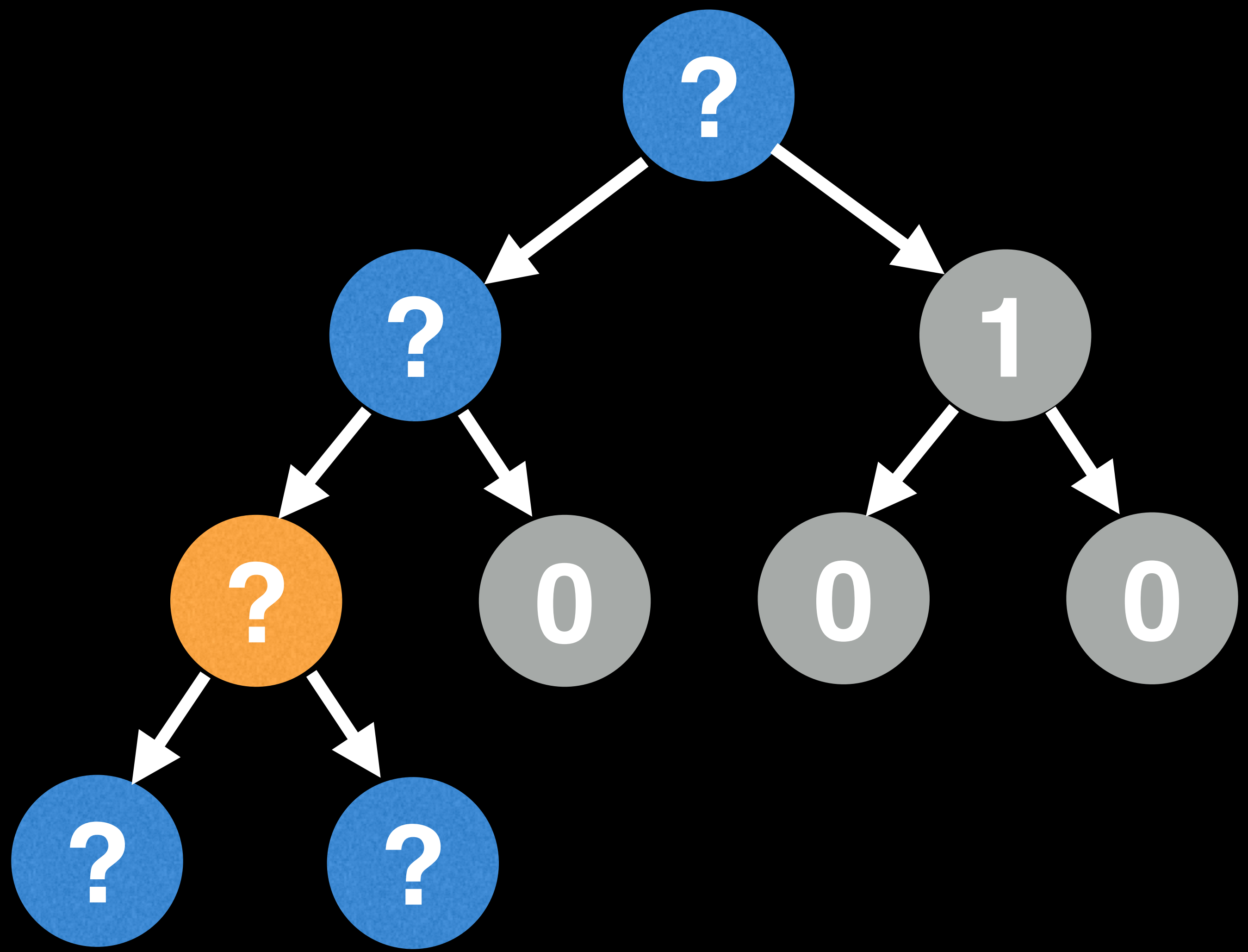


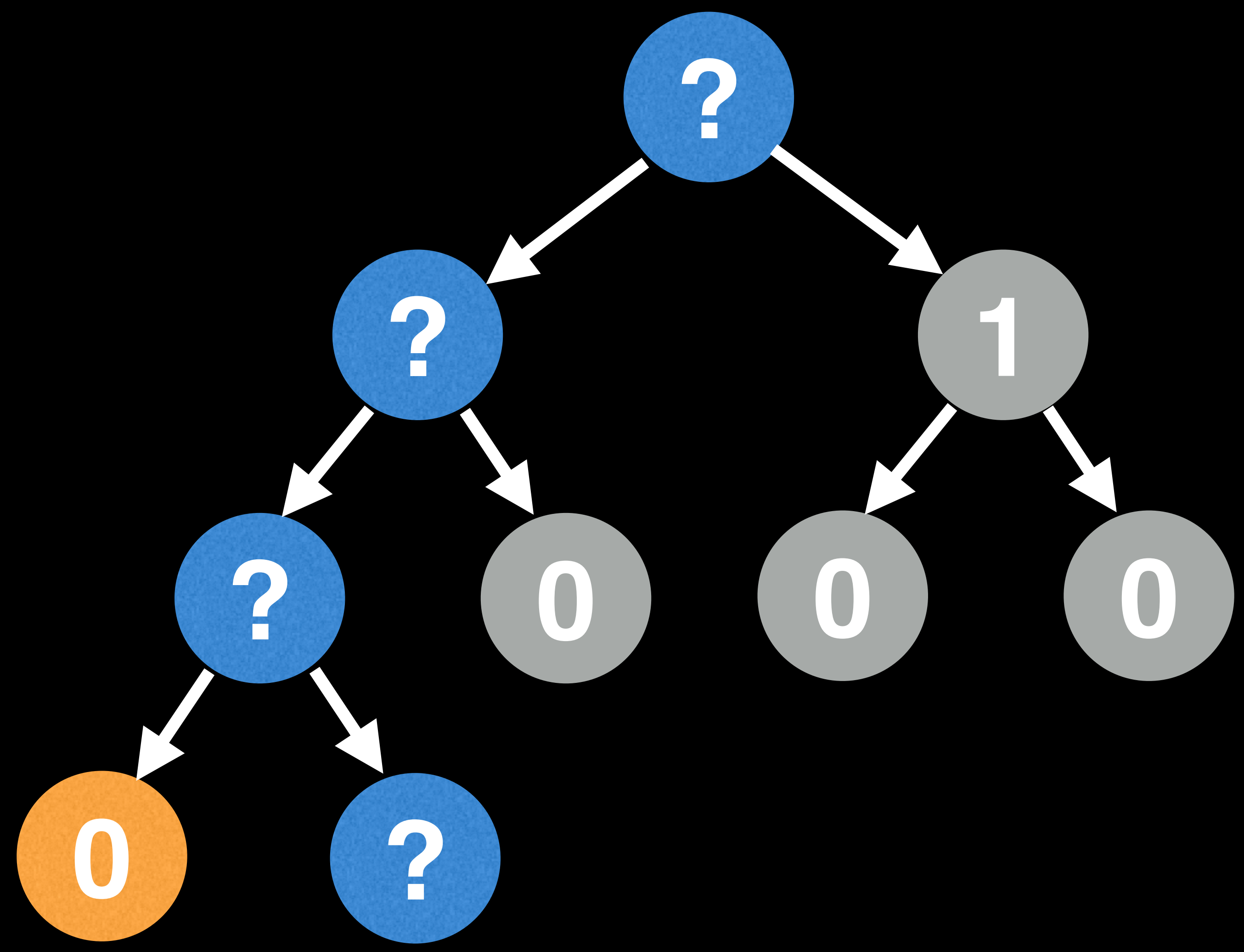
?
?
1
?
?
0
0
?
?



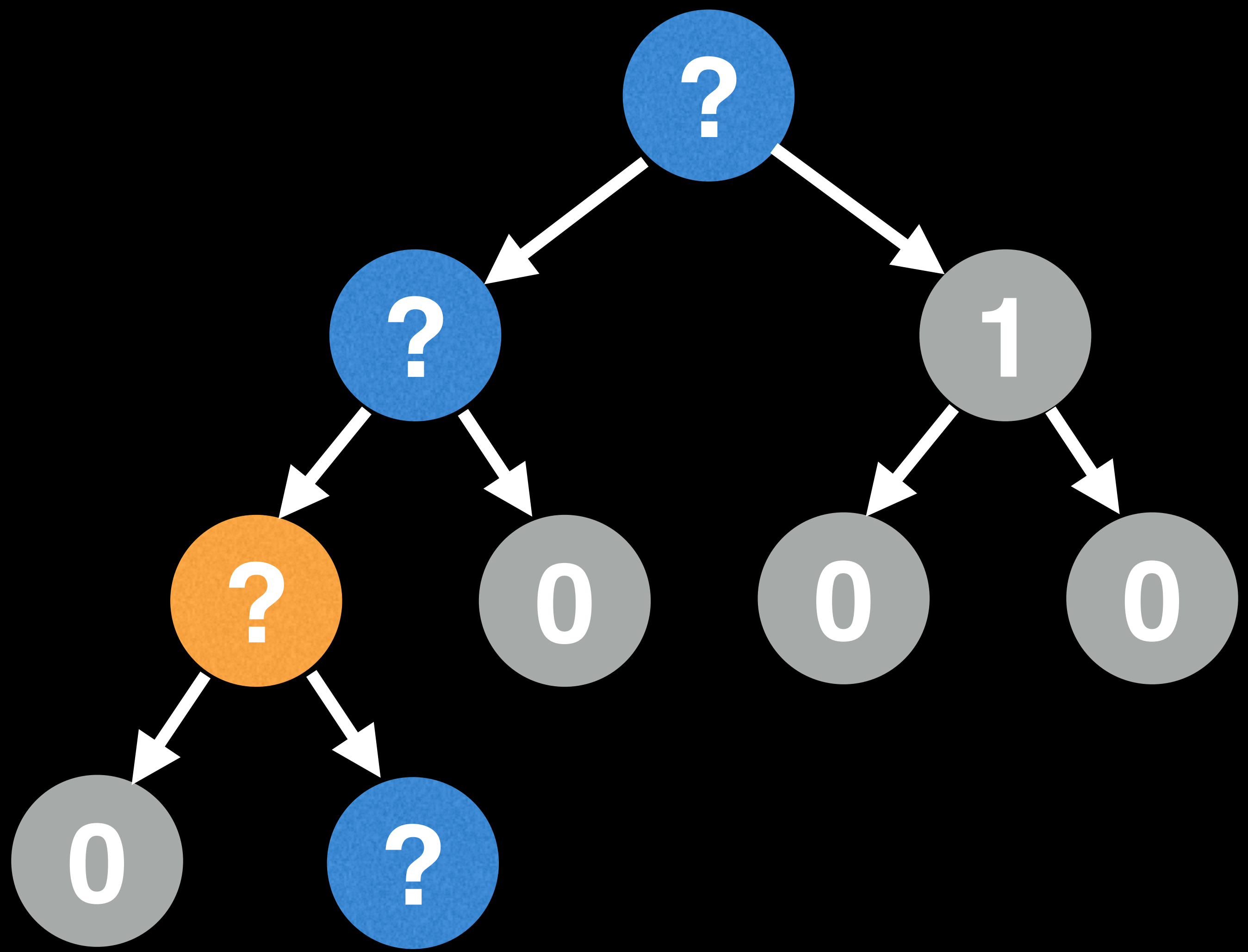
Leaf node has a height of 0

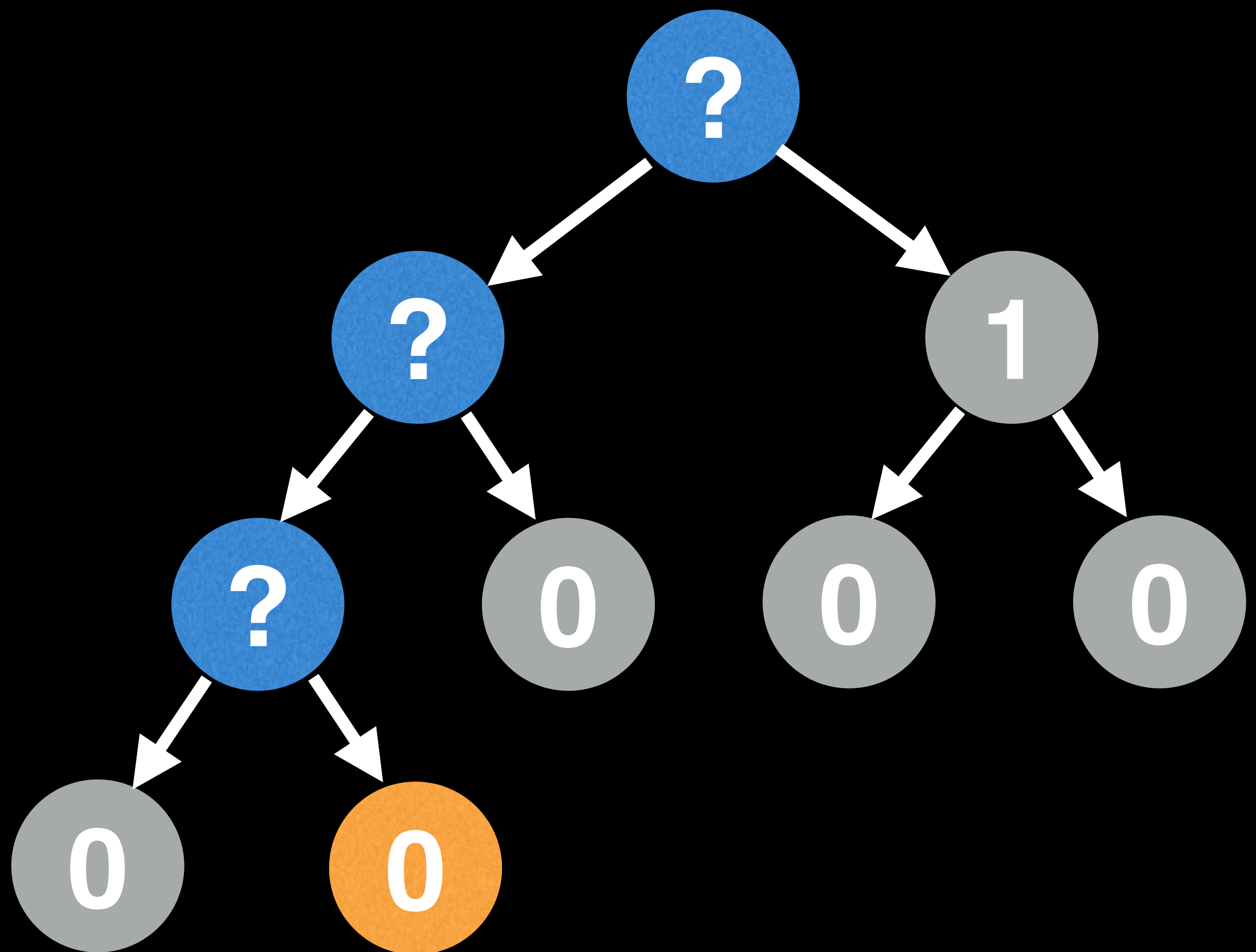

?
?
1
?
0
0
0
?
?


?
?
1
?
0
0
0
?
?

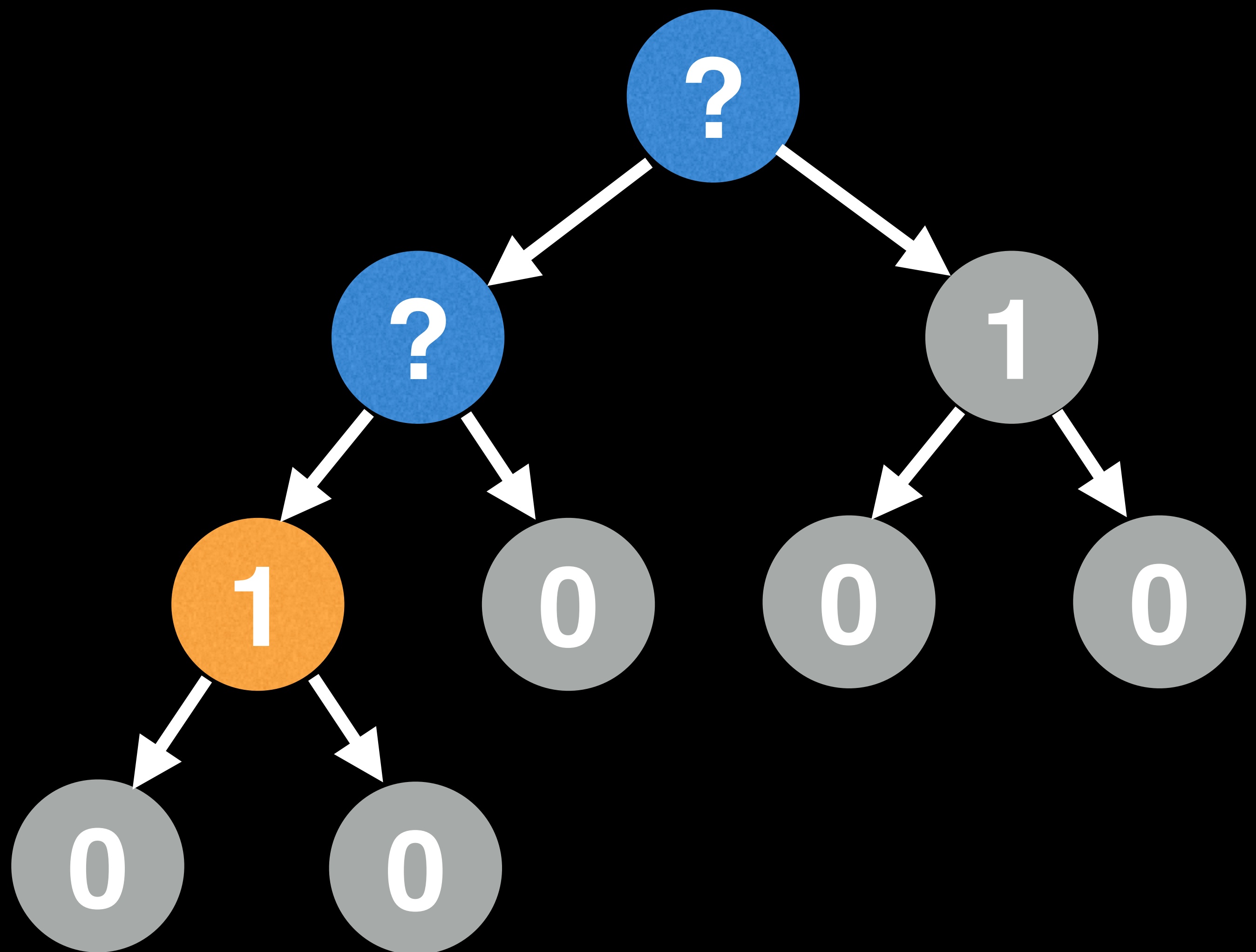


Leaf node has a height of 0

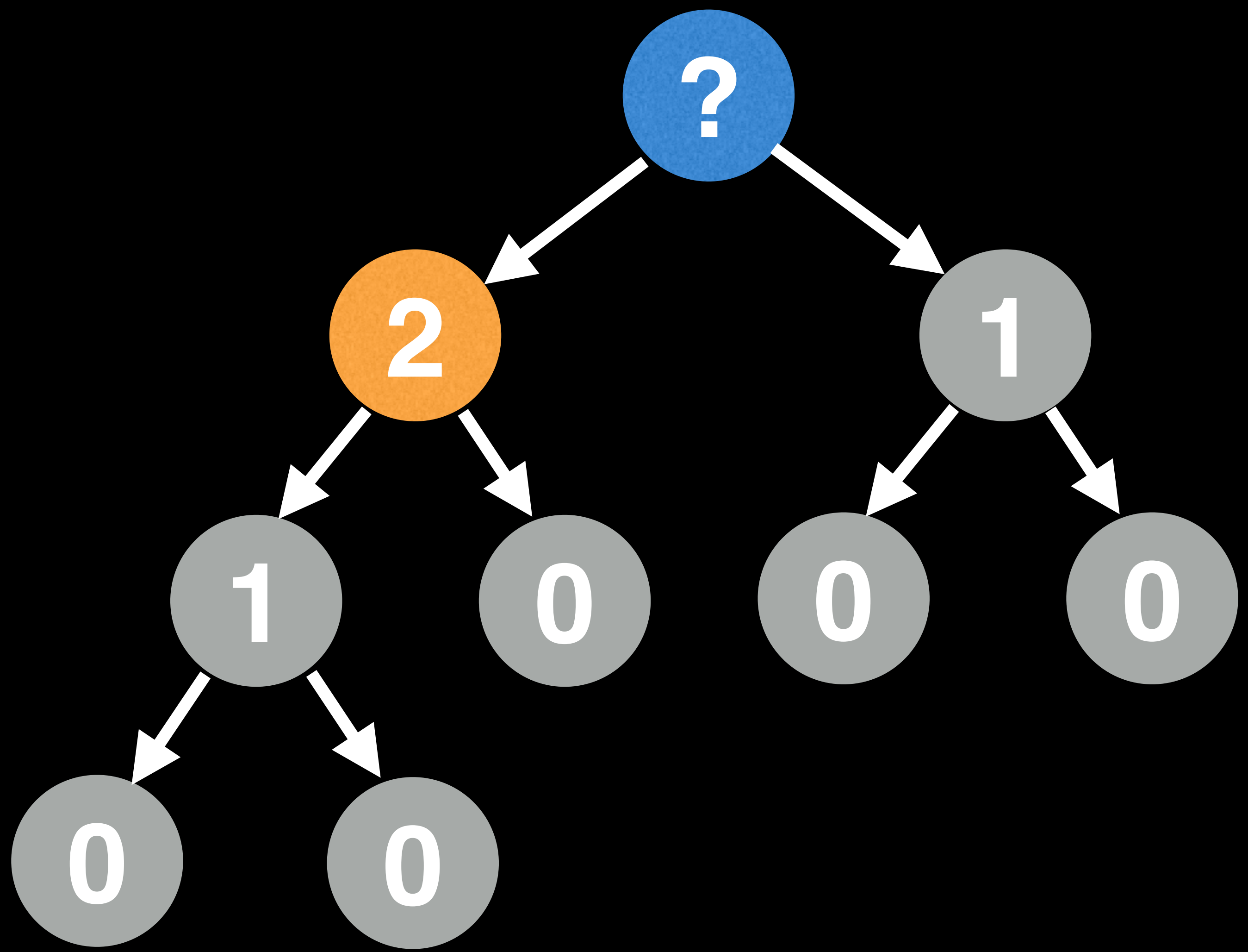

?
?
1
?
0
0
0
0
?



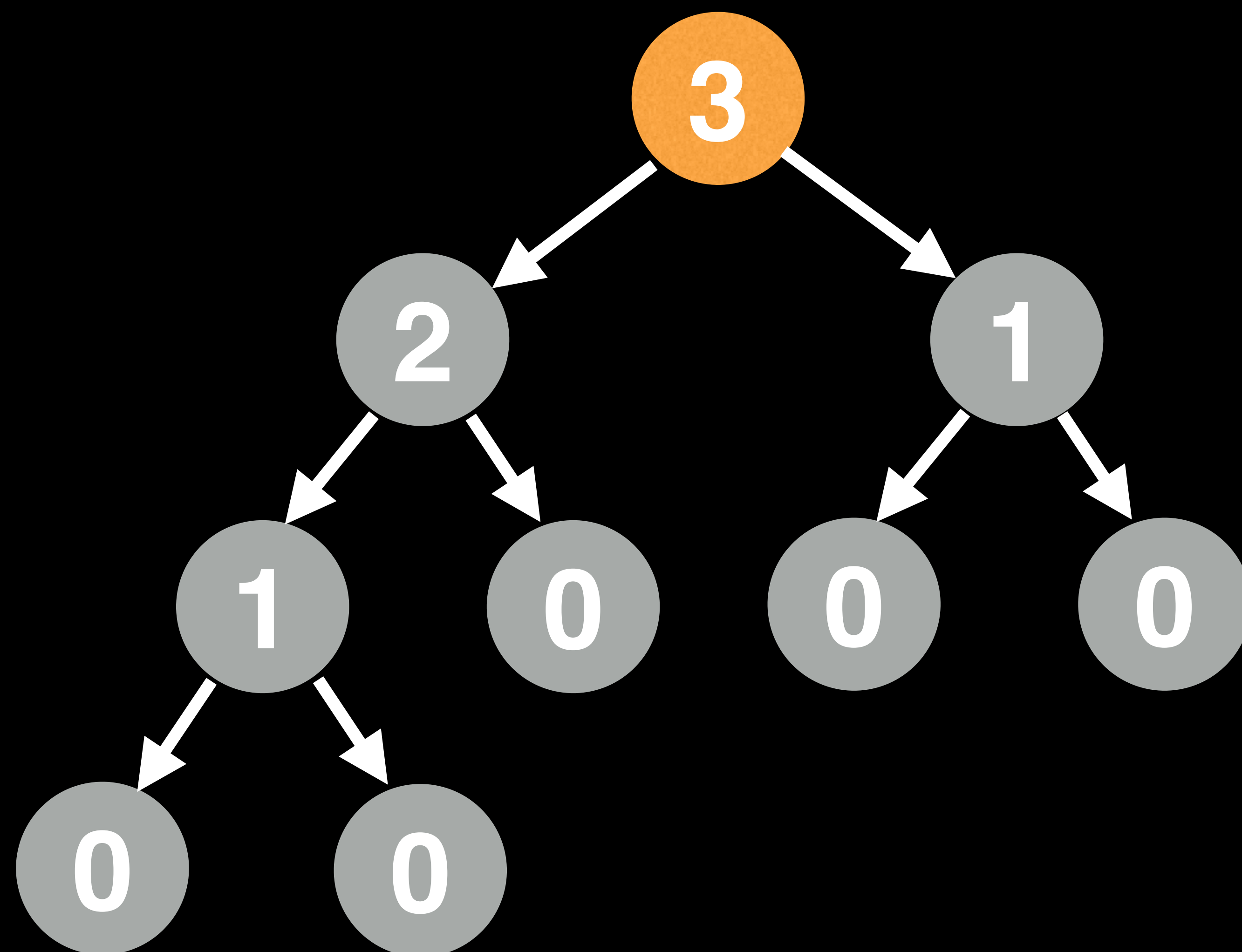
Leaf node has a height of 0



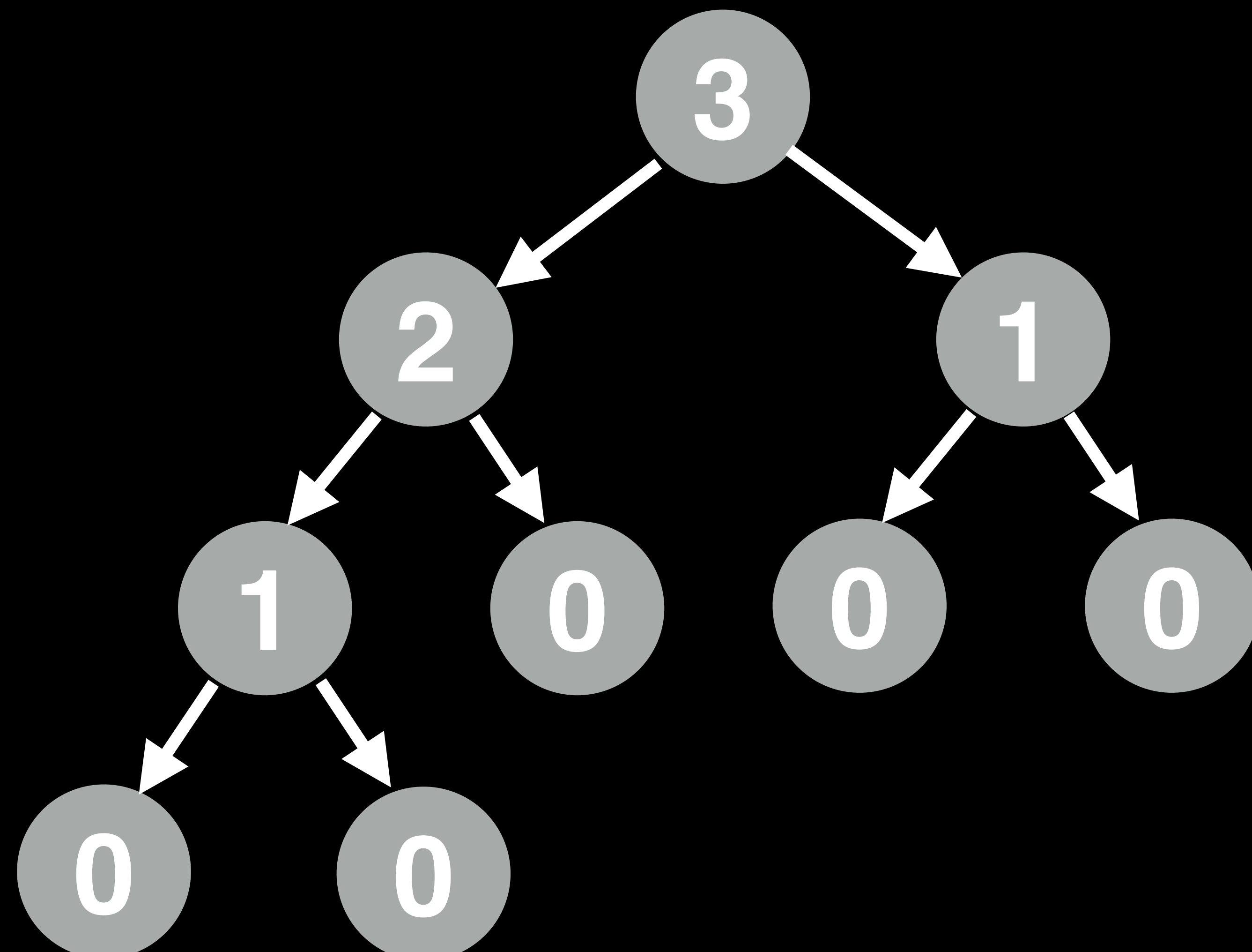
height = max(0, 0) + 1 = 1




```
height = max(1, 0) + 1 = 2
```



`height = max(2, 1) + 1 = 3`


3
2
1
1
0
0
0
0
0

```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
     return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
     return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
     return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

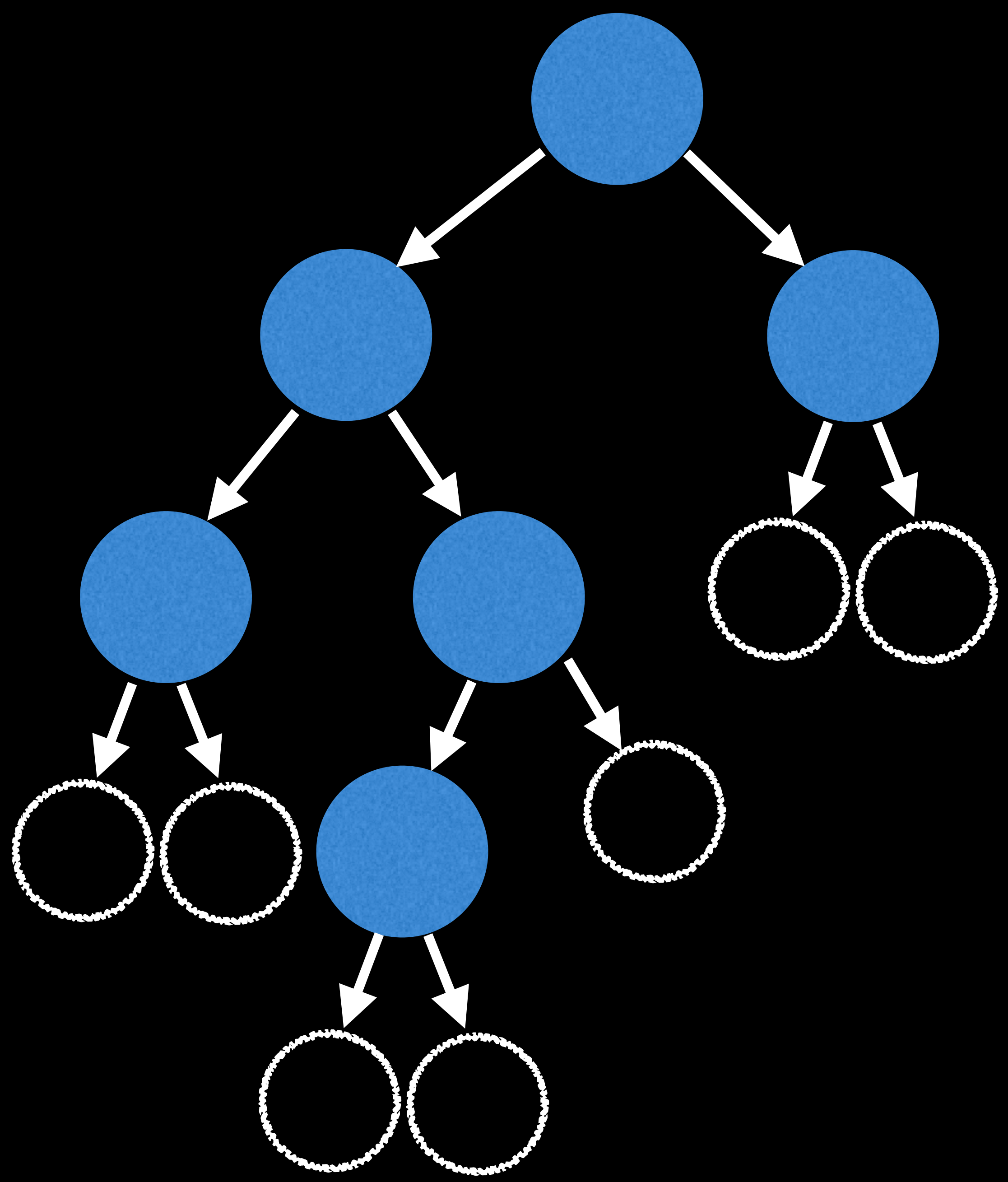
```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
     return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
     return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

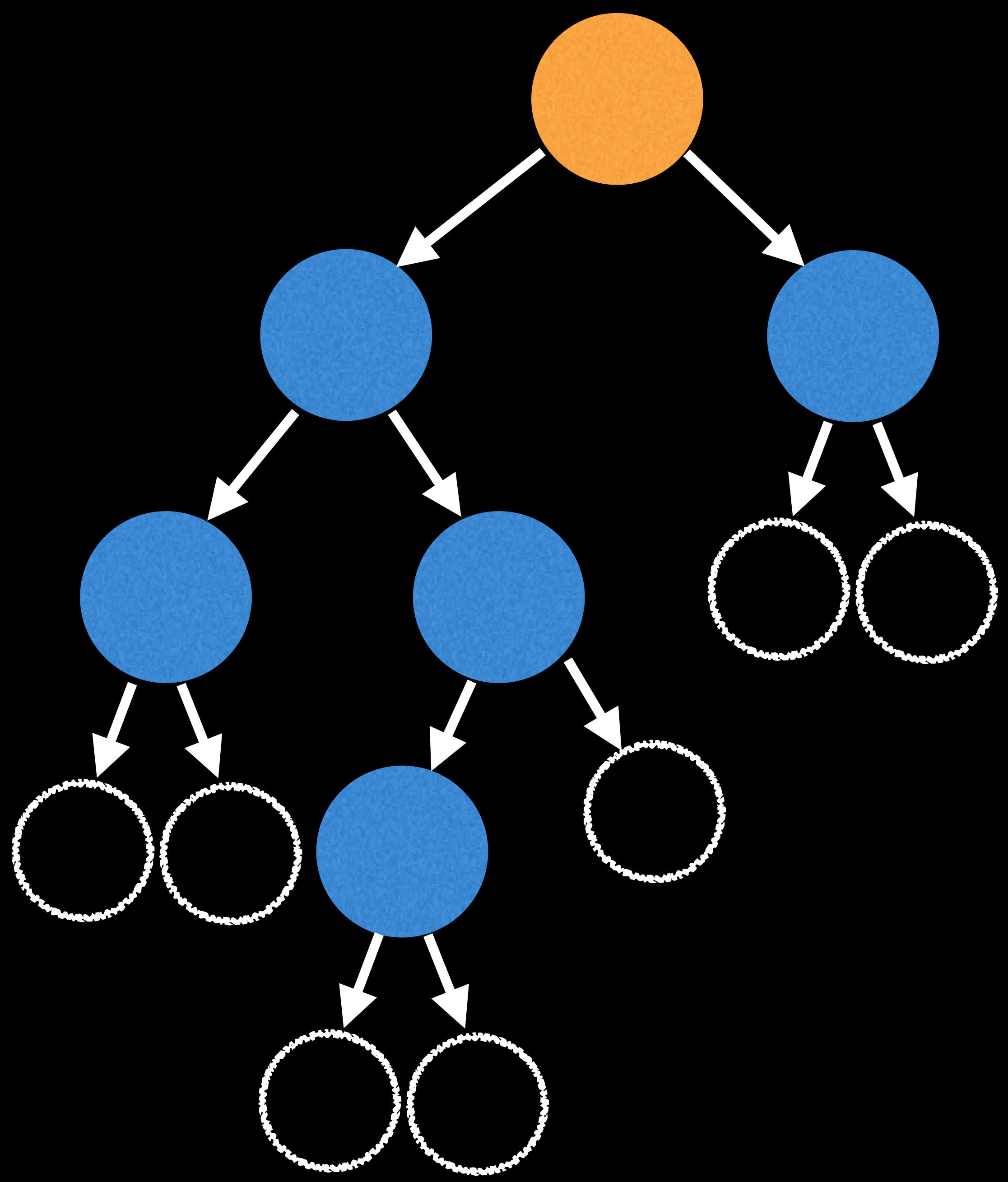
```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
     return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

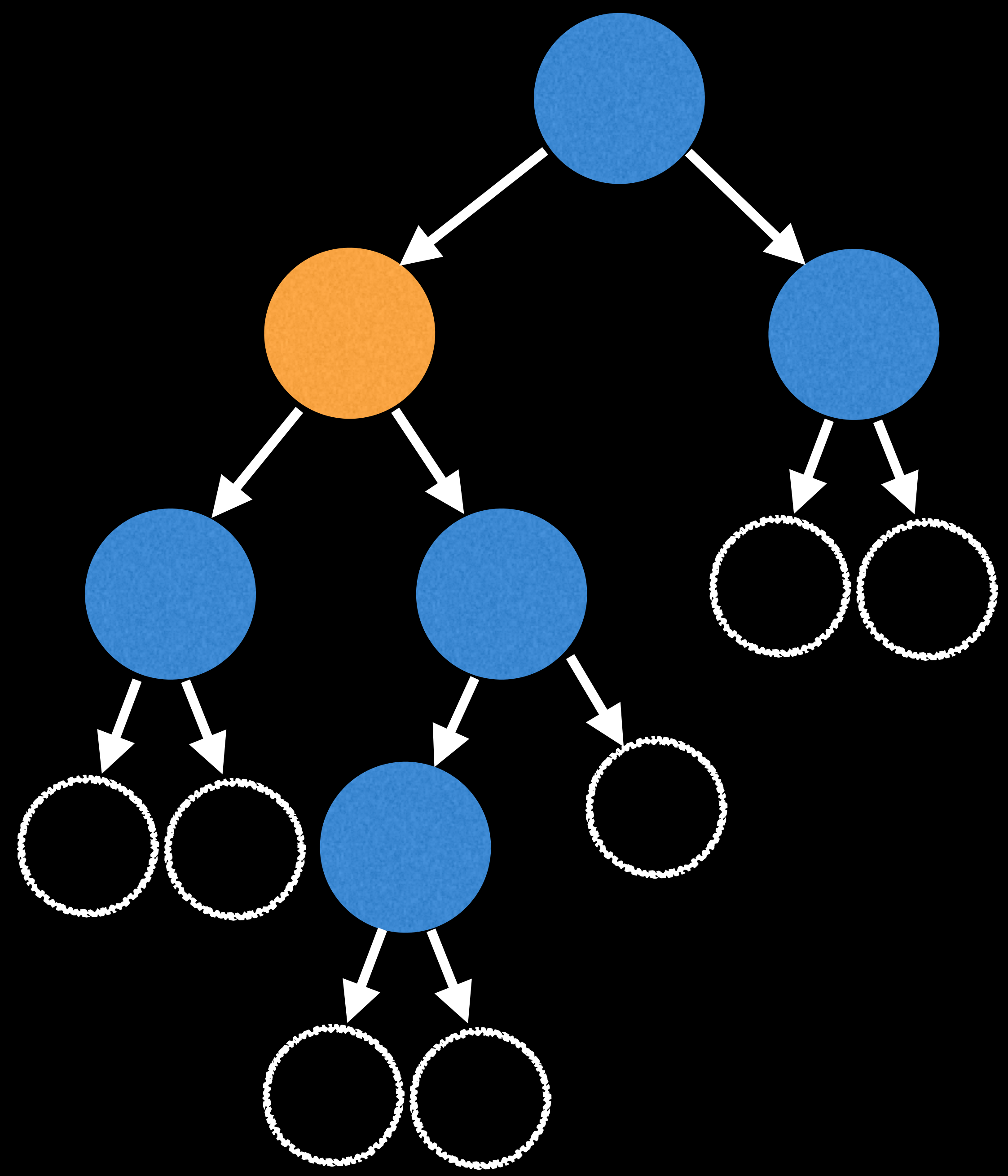
```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Return -1 when we hit a null node
  # to correct for the right height.
  if node == null:
     return -1

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

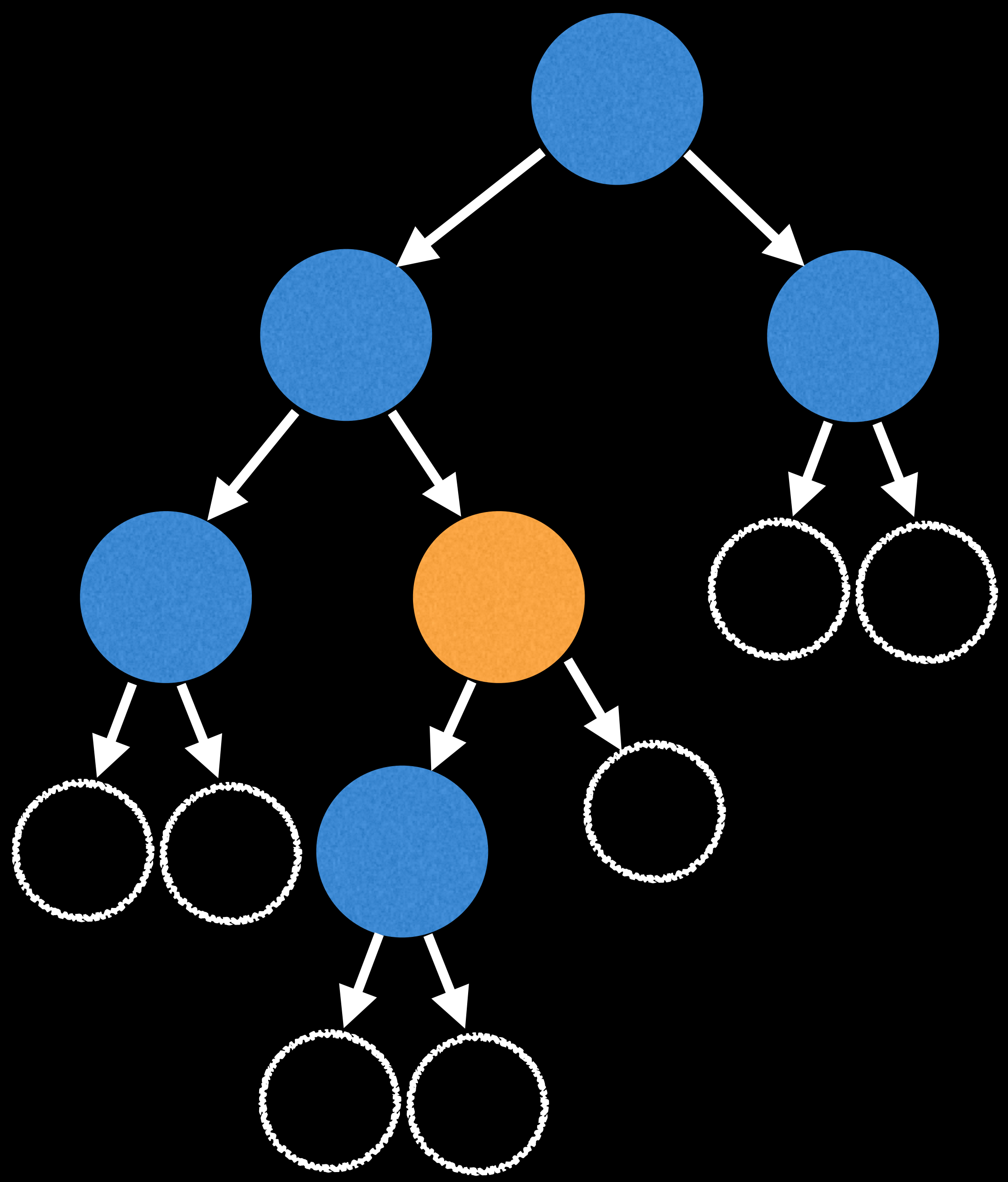
Notice that if we visit the null nodes our tree is one unit taller.

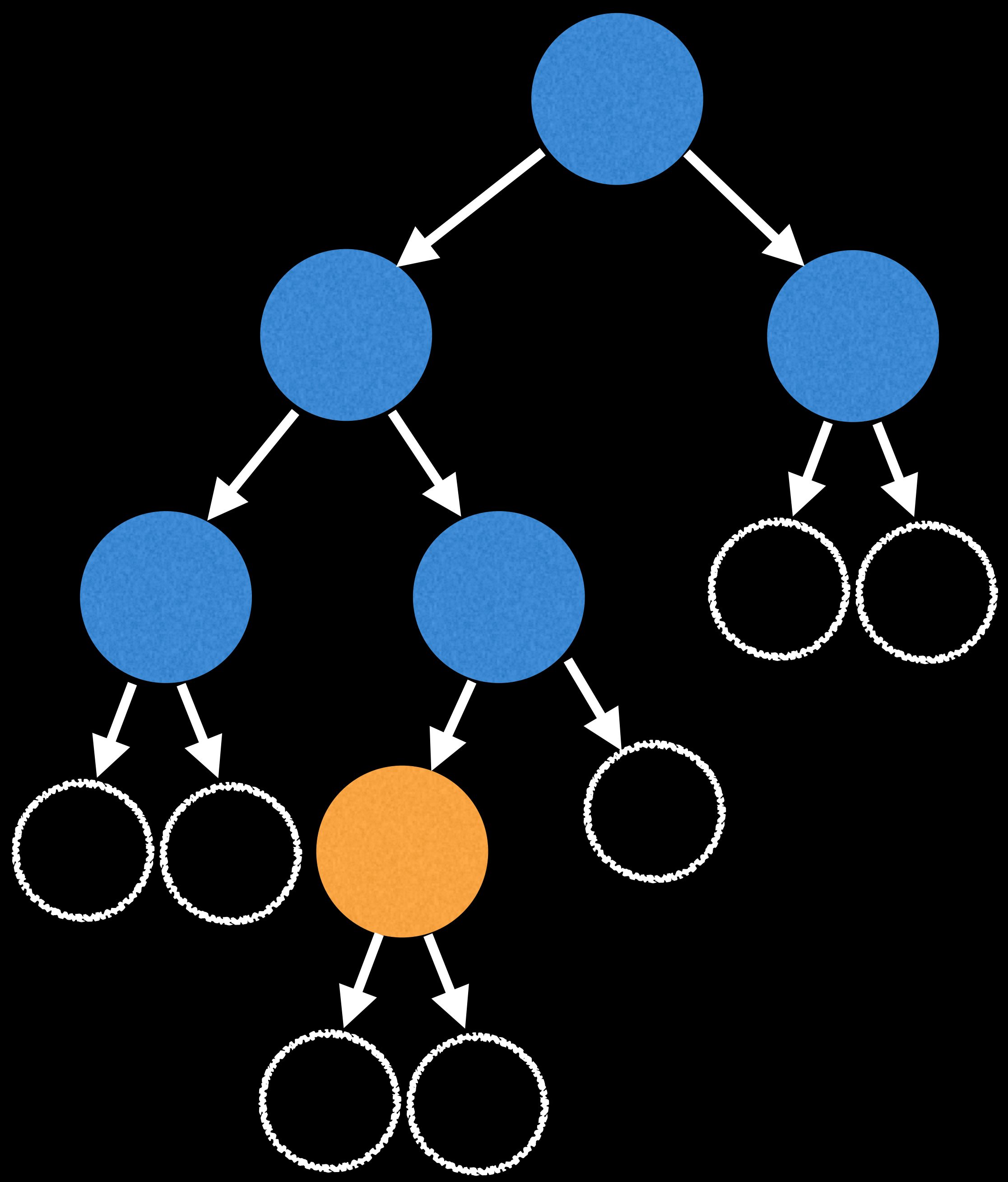


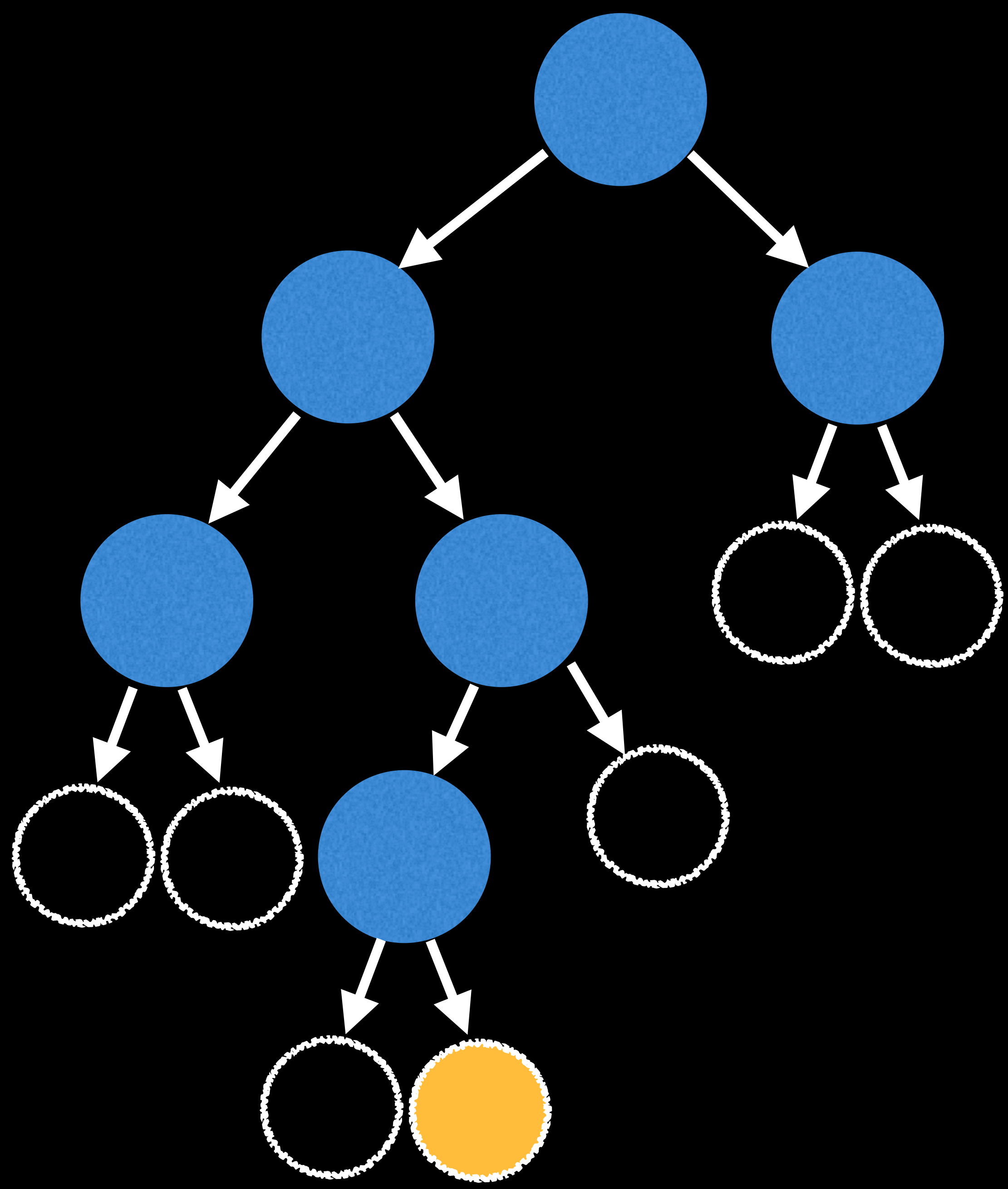
When we go down the tree we need to correct for the height added by the null nodes.

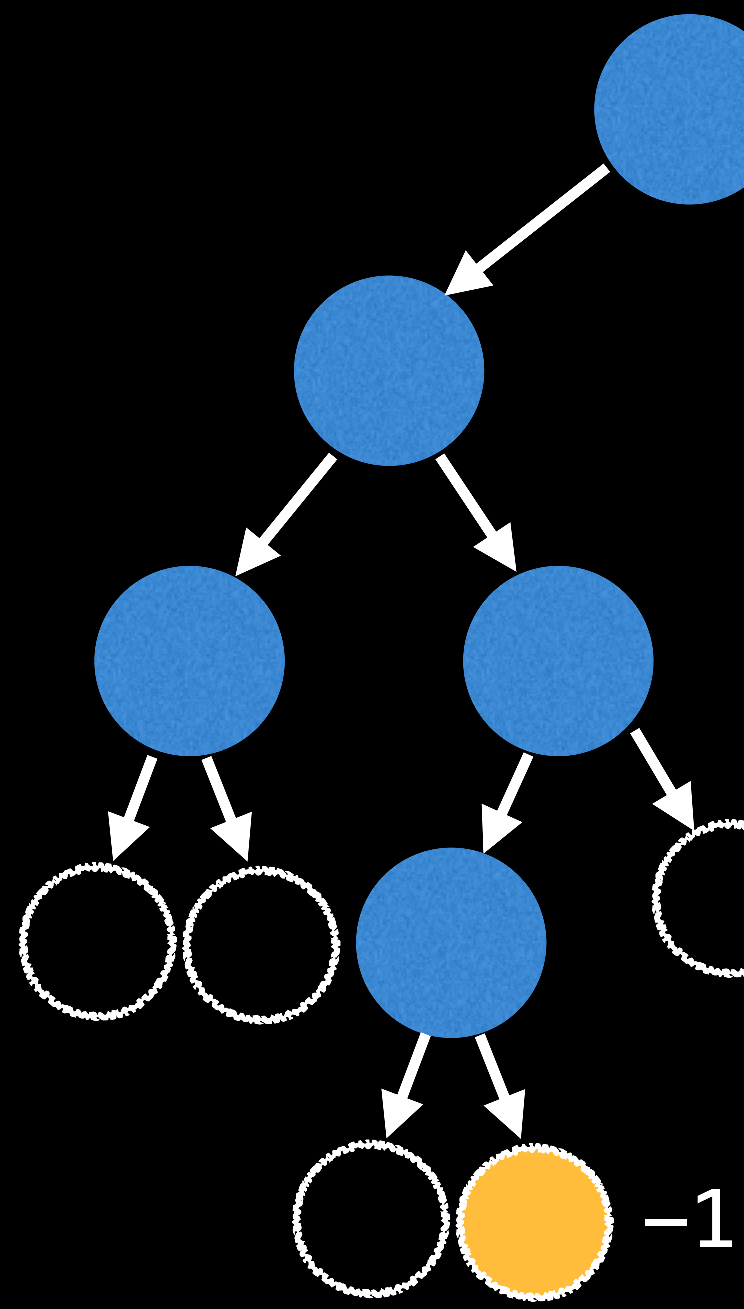


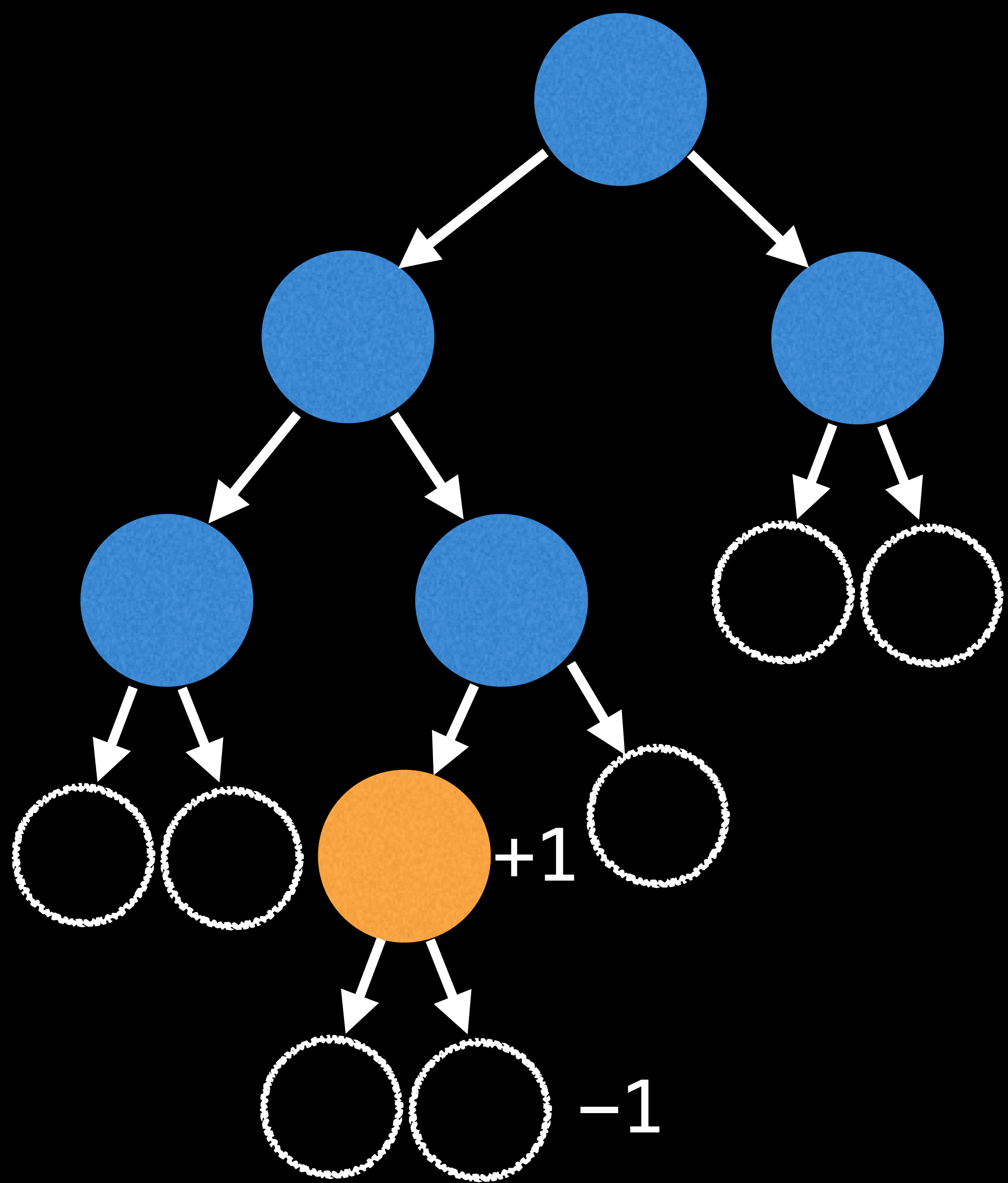


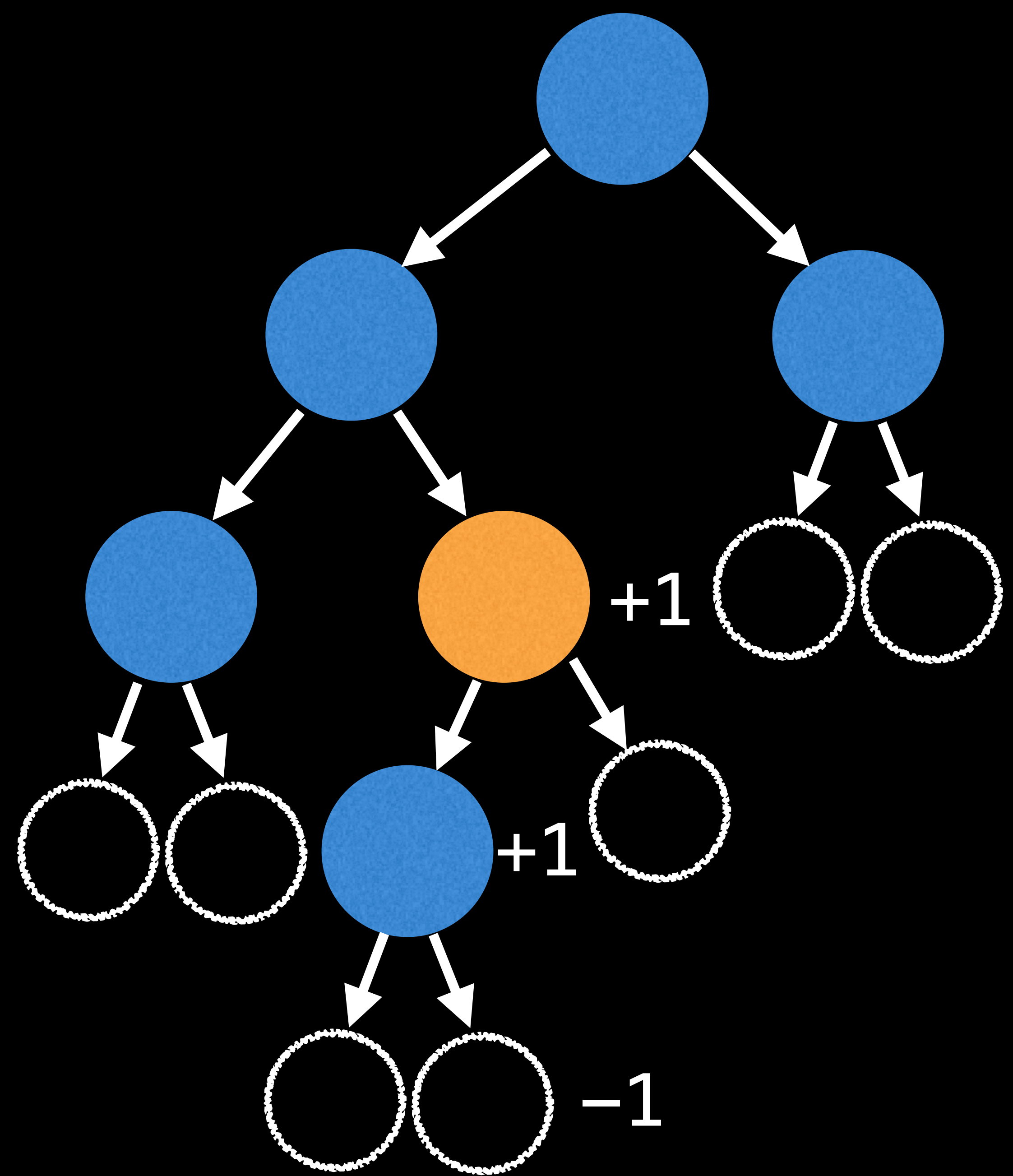


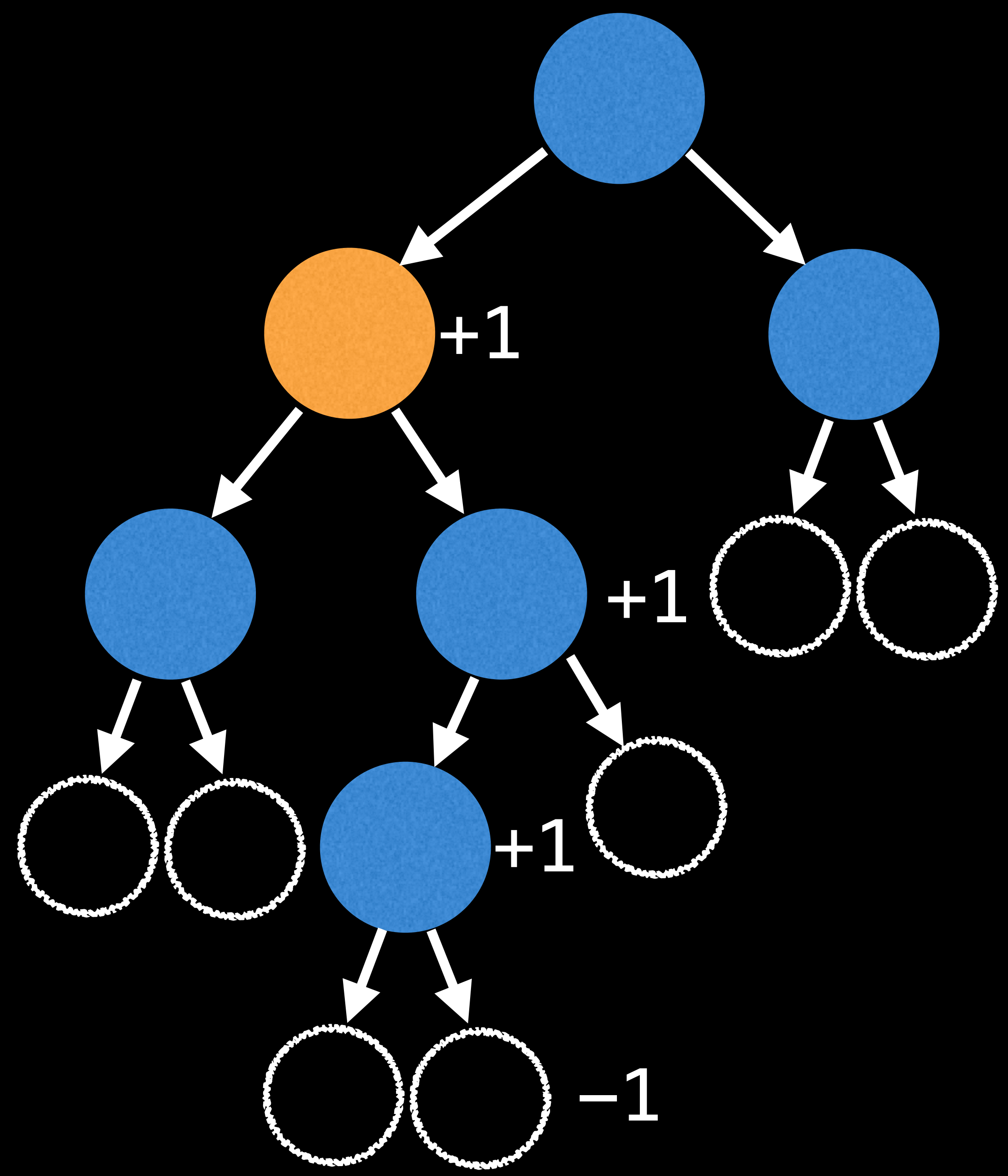


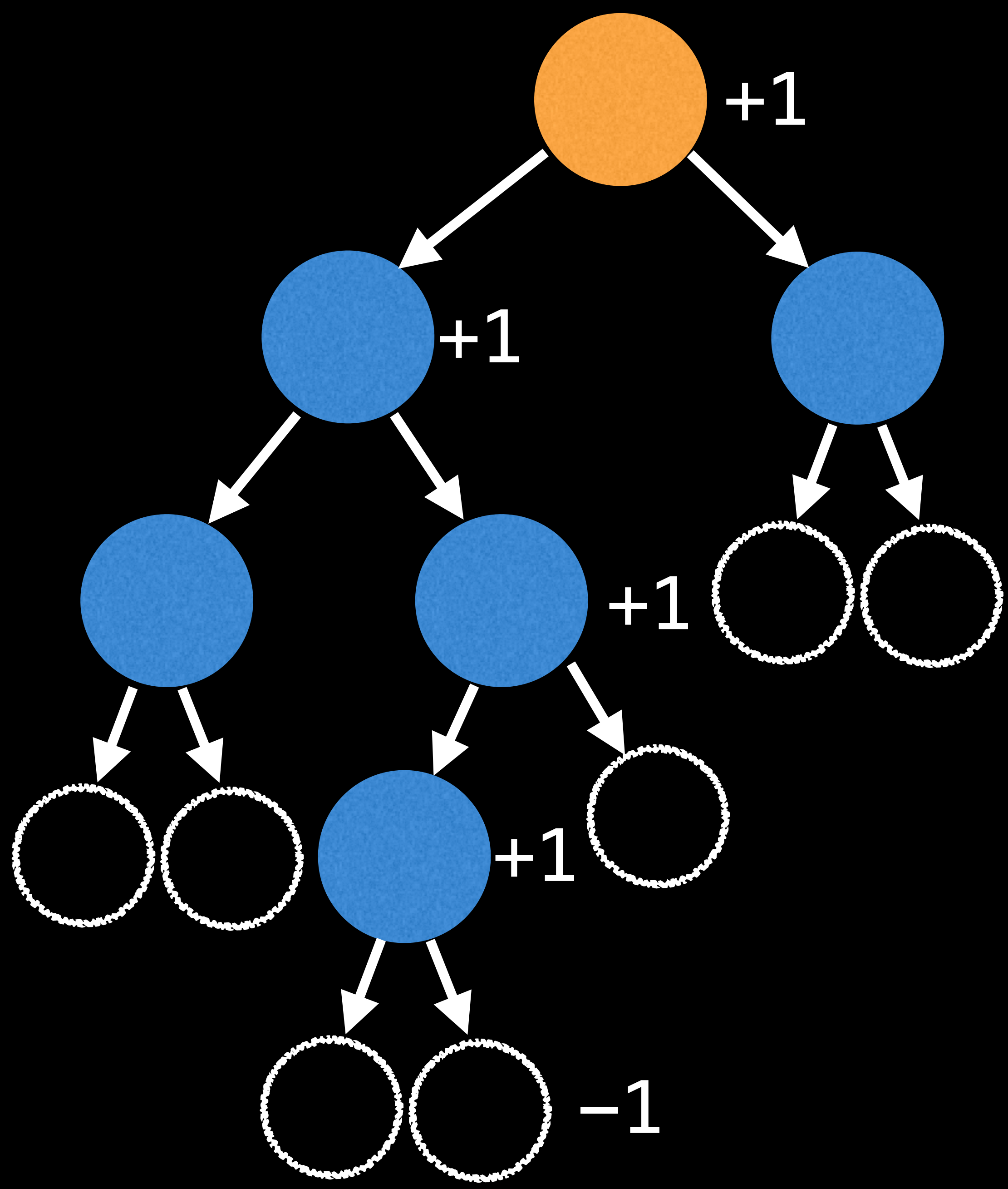


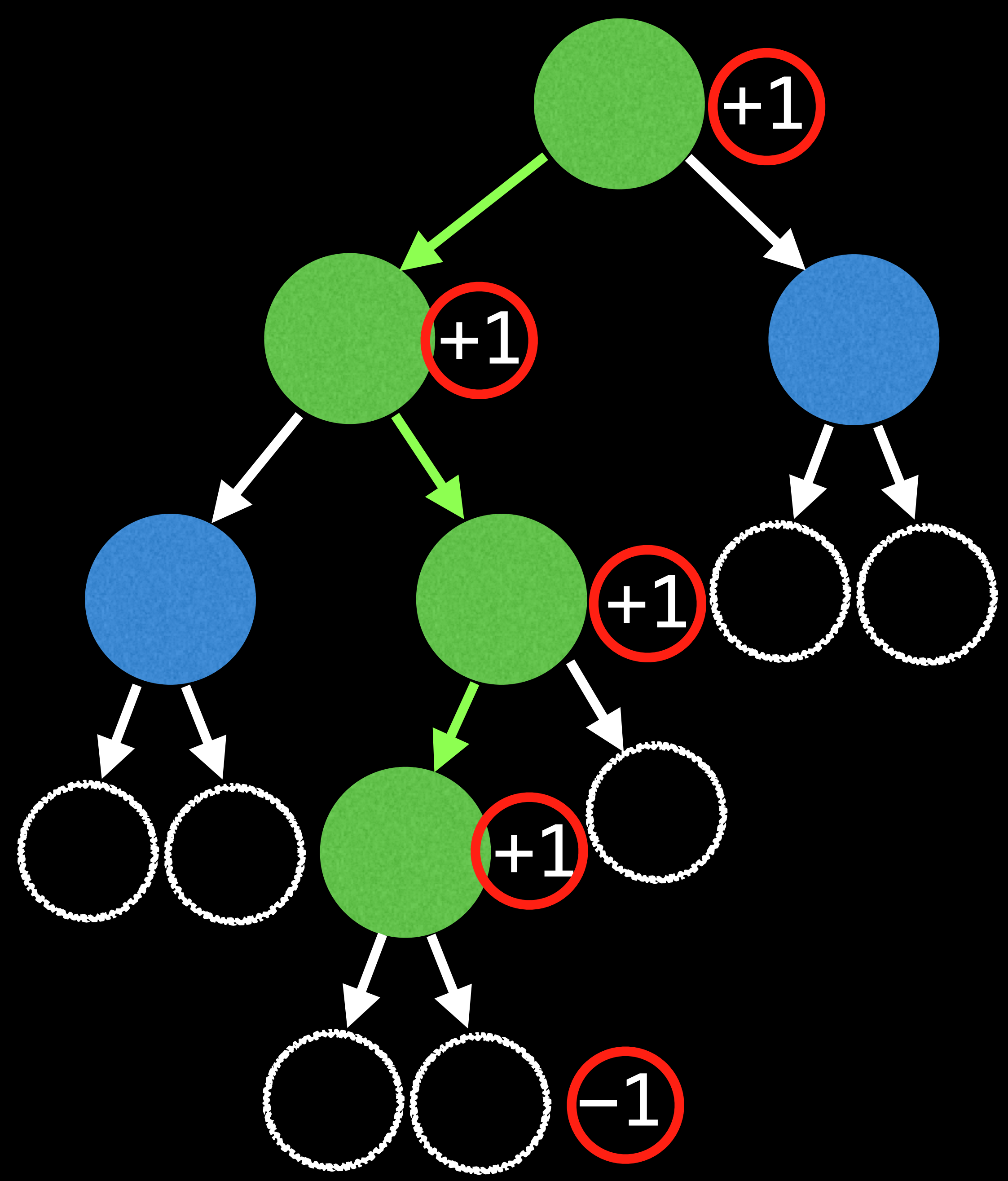

−1


+1
−1


+1
+1
−1


+1
+1
+1
−1


+1
+1
+1
+1
−1

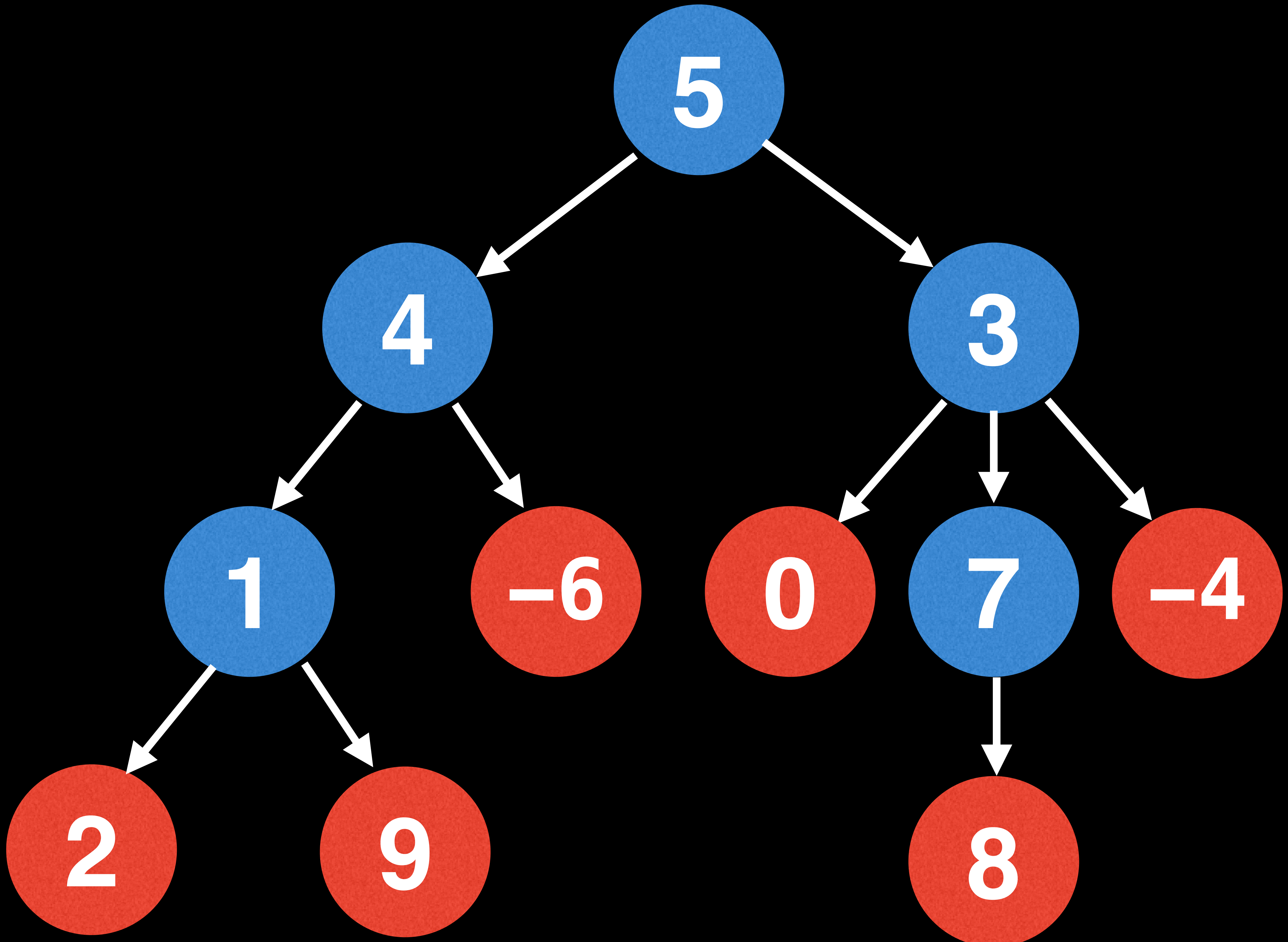


$1 + 1 + 1 + 1 - 1 = 3$

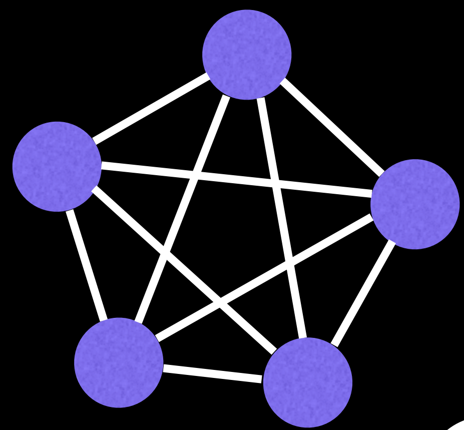
```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Return -1 when we hit a null node
  # to correct for the right height.
  if node == null:
     return -1

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

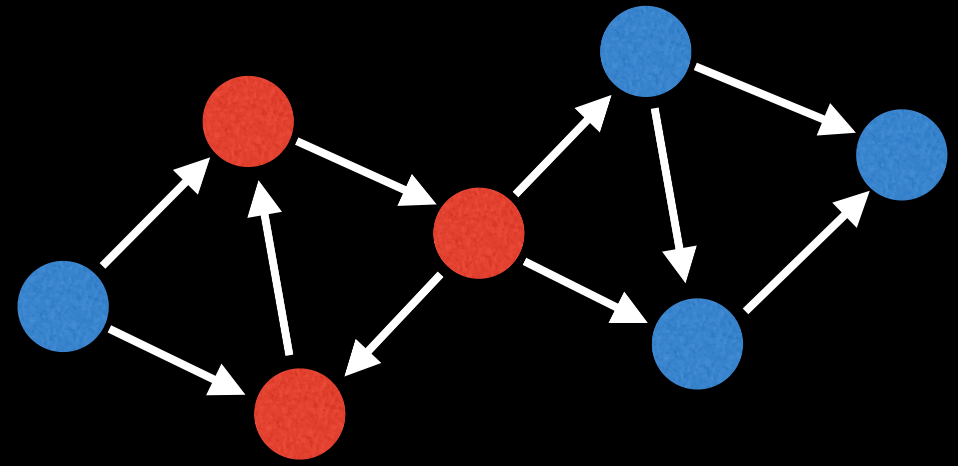
Next Video: rooting a tree

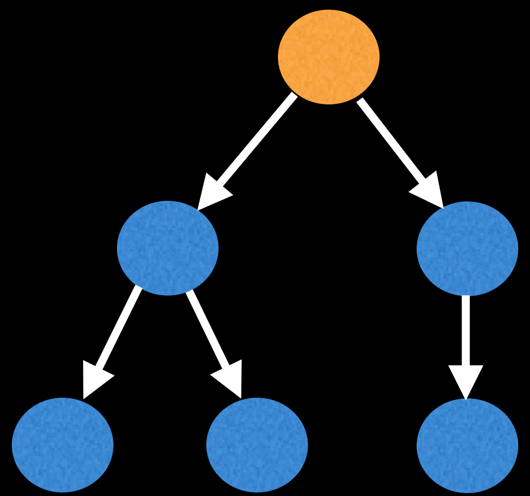
Beginner Tree Algorithms



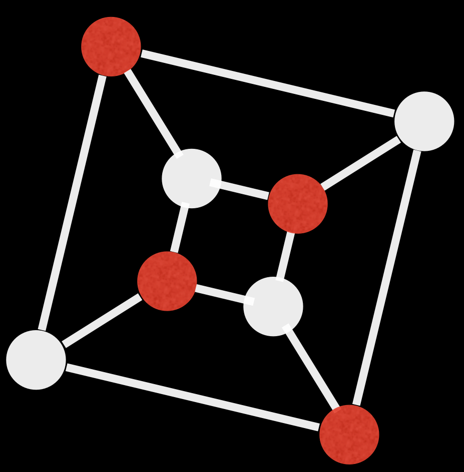
2 + 9 − 6 + 0 + 8 − 4 = 9

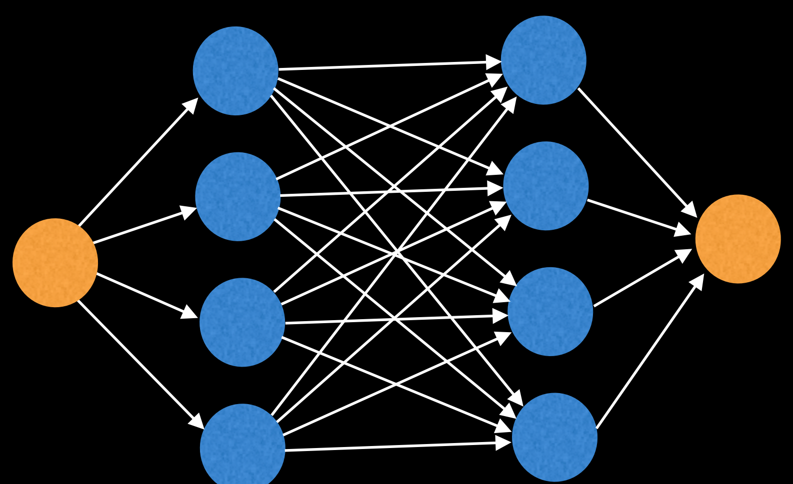


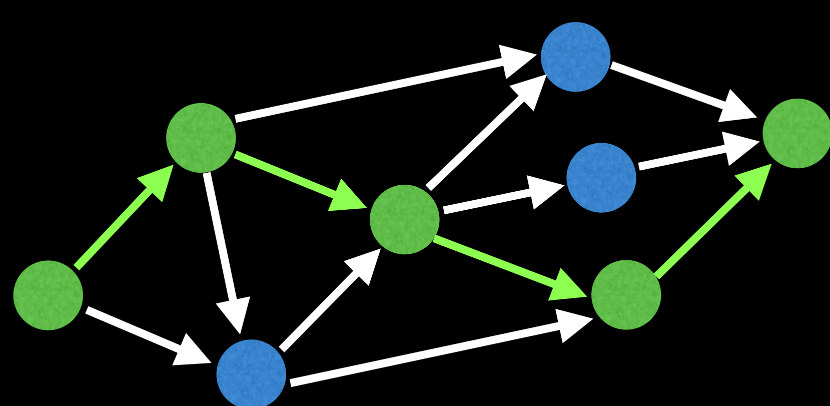




Graph Theory Video Series





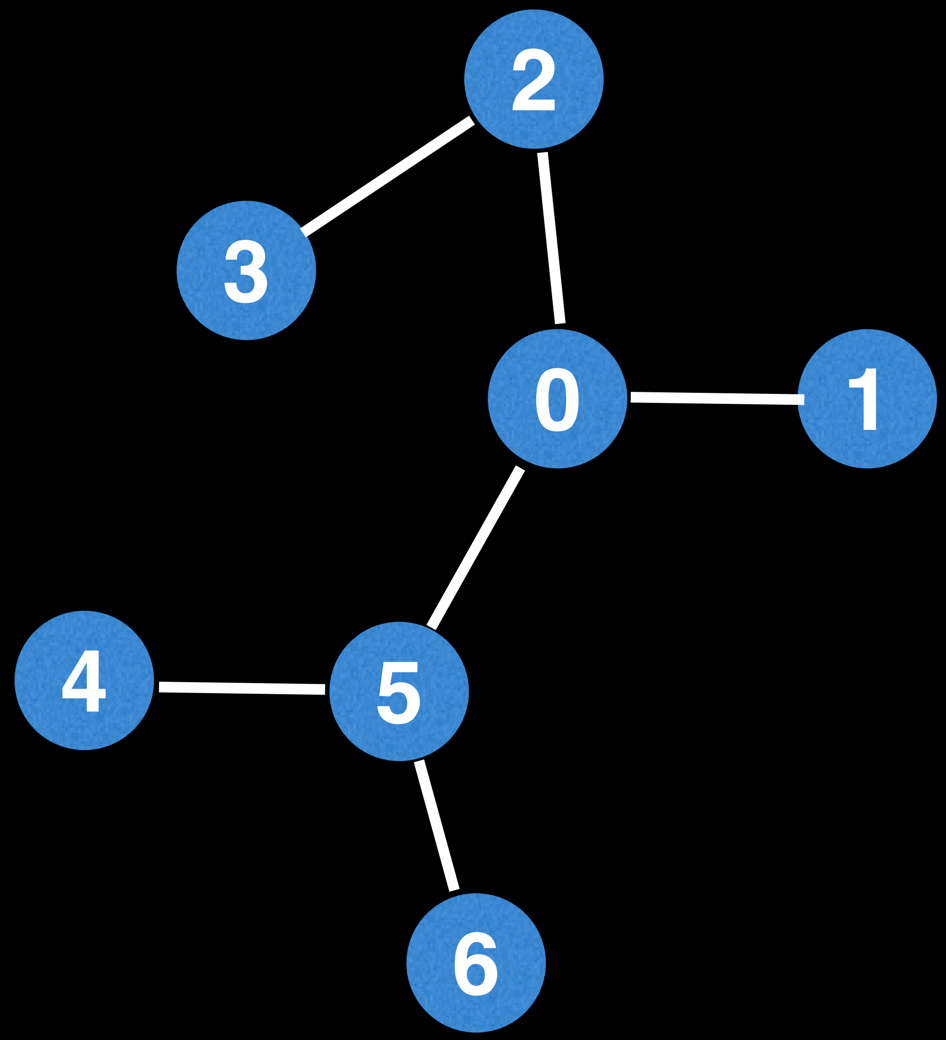


Rooting a tree

William Fiset

Rooting a tree

Sometimes it's useful to root an undirected tree to add structure to the problem you're trying to solve.

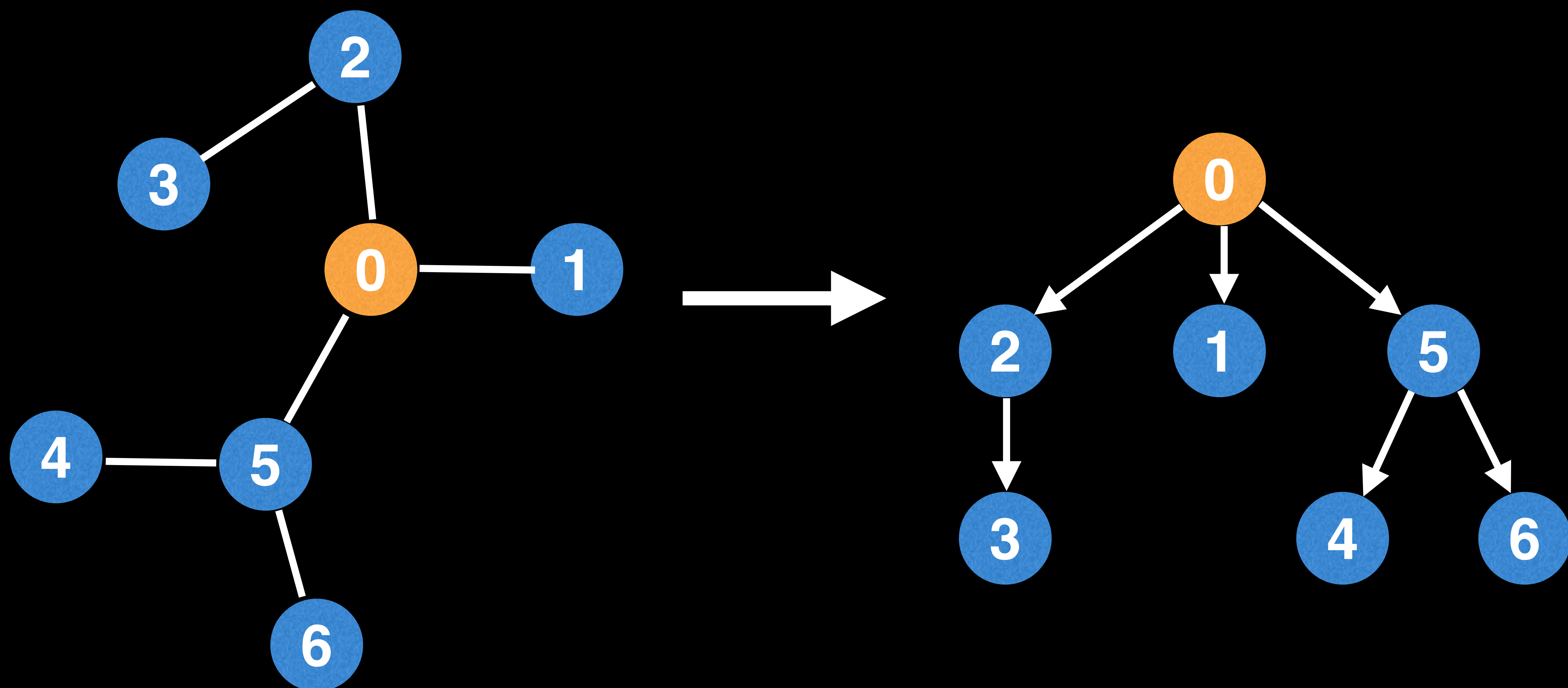


Undirected graph adjacency list:

0 -> [2, 1, 5]
1 -> [0]
2 -> [3, 0]
3 -> [2]
4 -> [5]
5 -> [4, 6, 0]
6 -> [5]

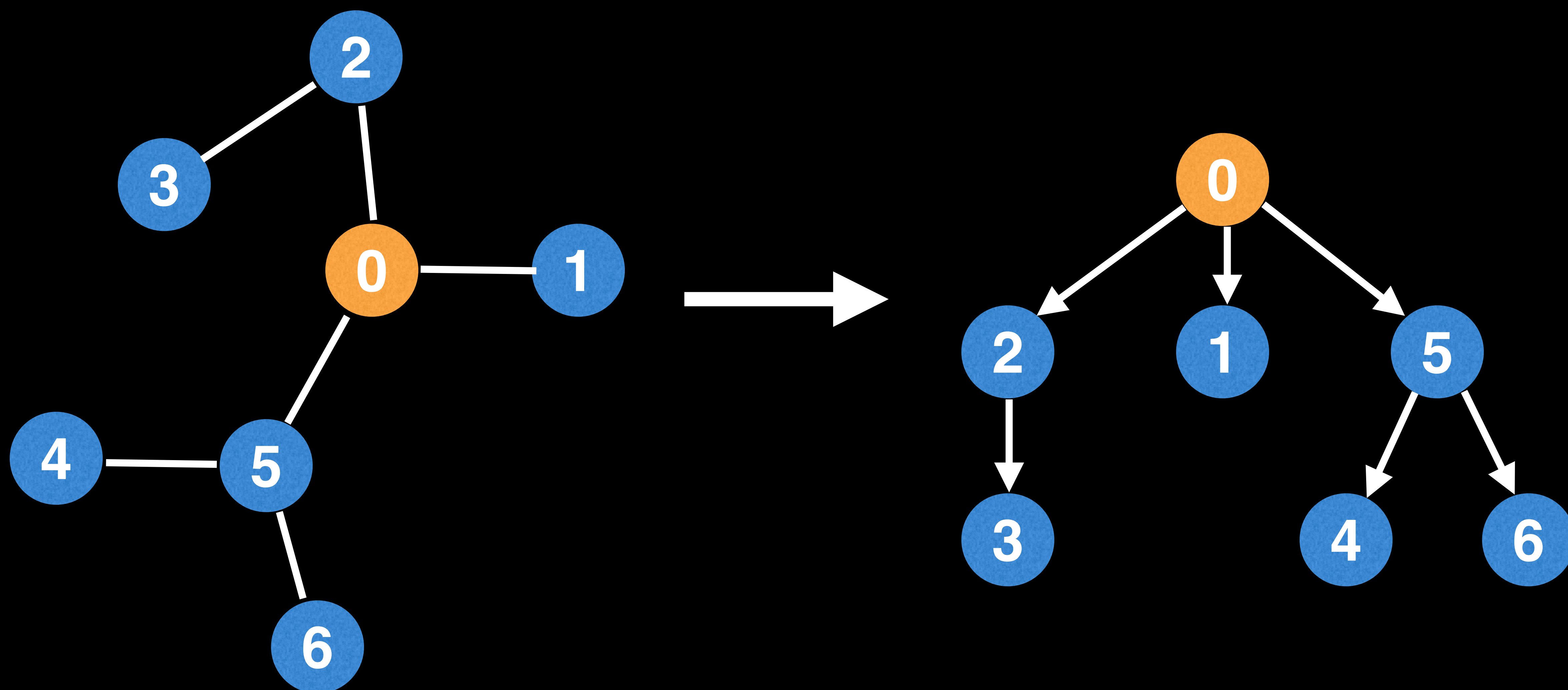
Rooting a tree

Sometimes it's useful to root an undirected tree to add structure to the problem you're trying to solve.



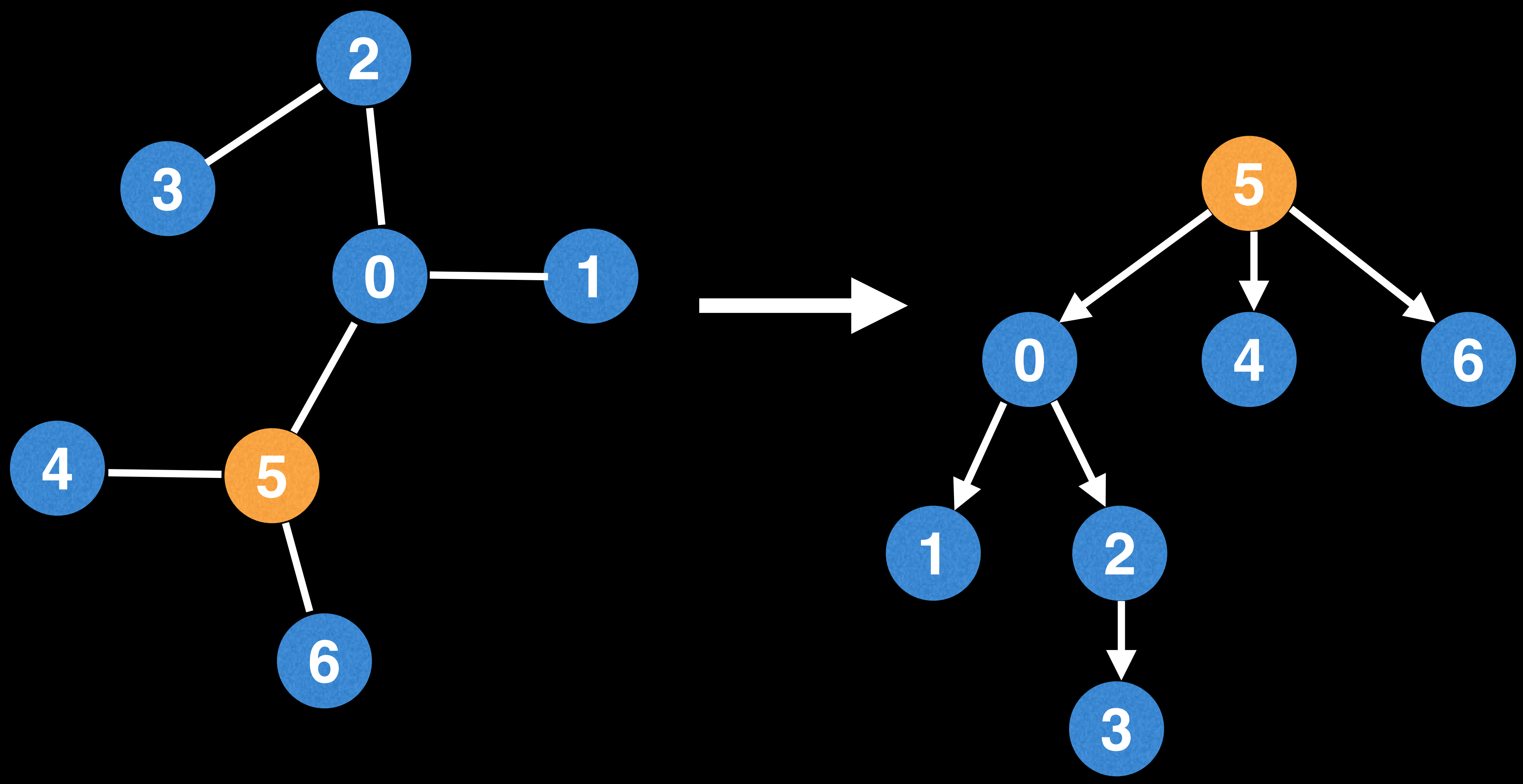
Rooting a tree

Conceptually this is like "picking up" the tree by a specific node and having all the edges point downwards.

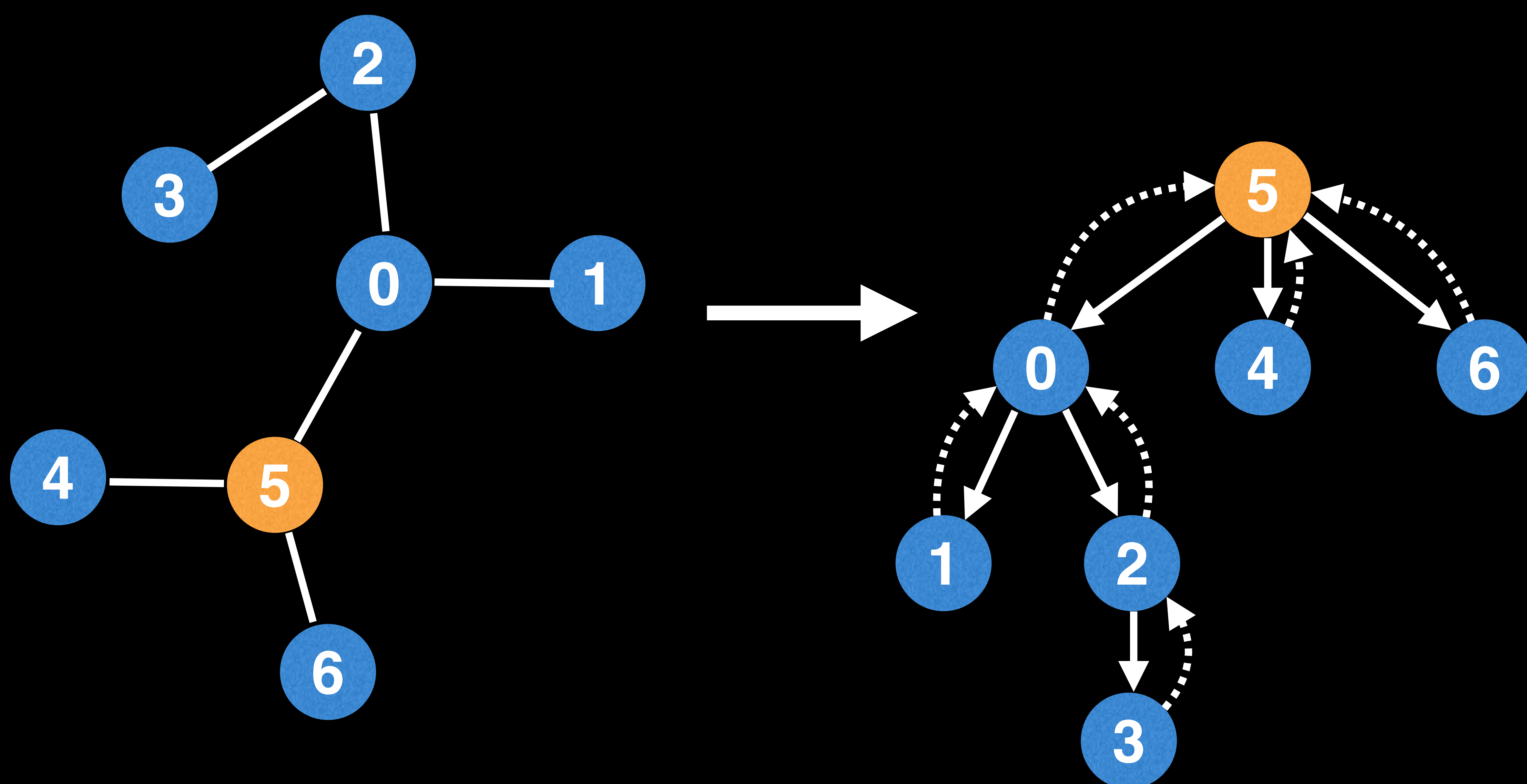


Rooting a tree

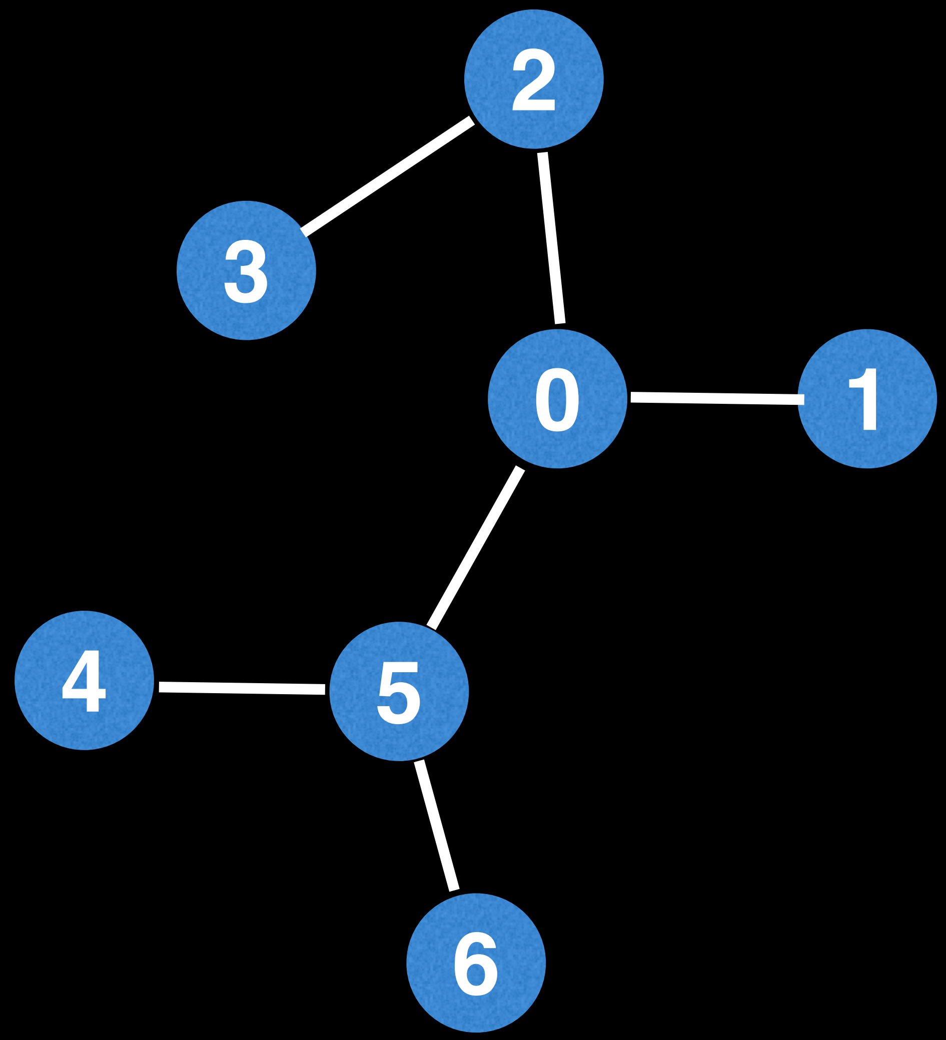
You can root a tree using any of its nodes.



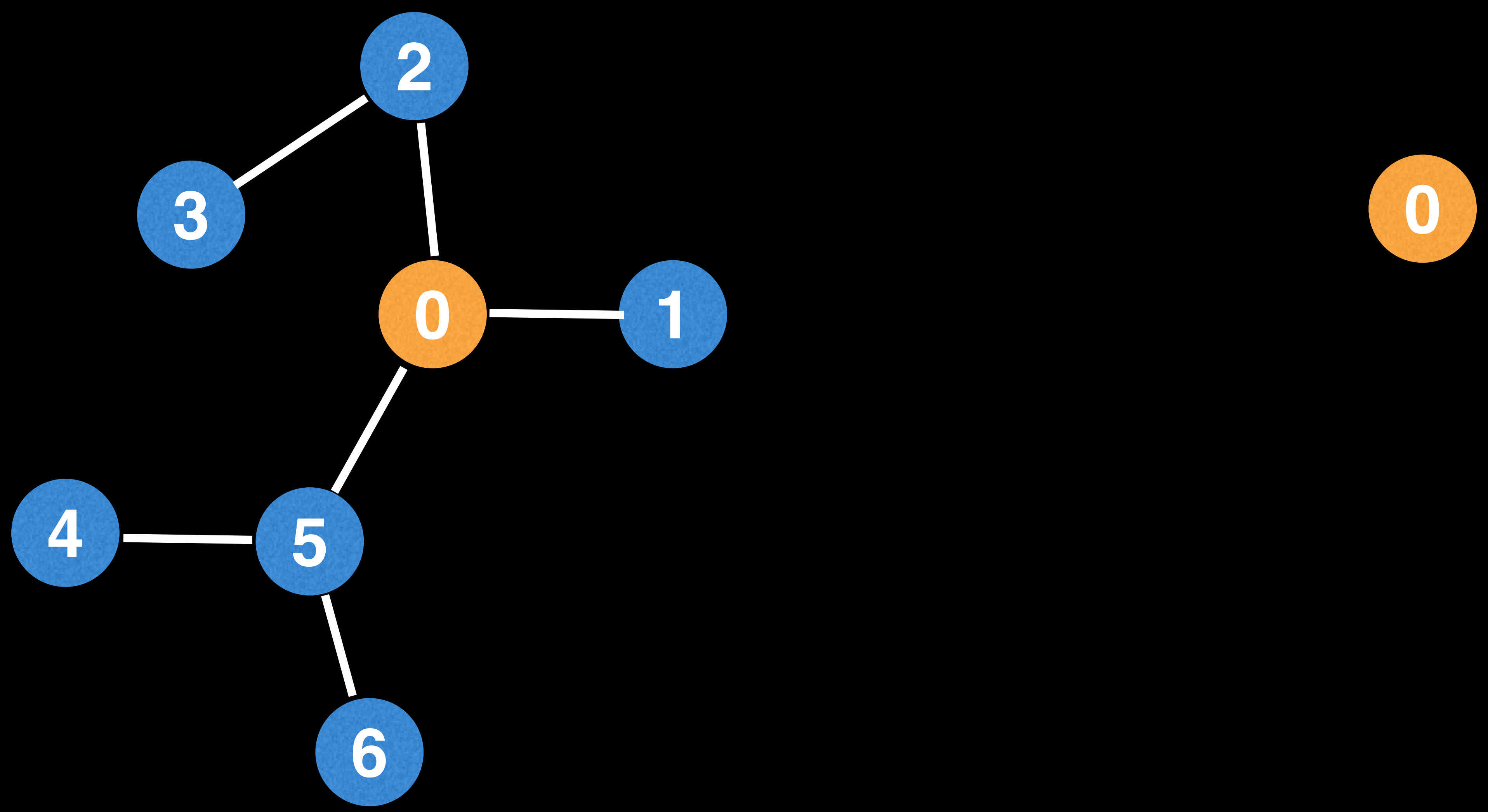
In some situations it's also useful to keep have a reference to the parent node in order to walk up the tree.



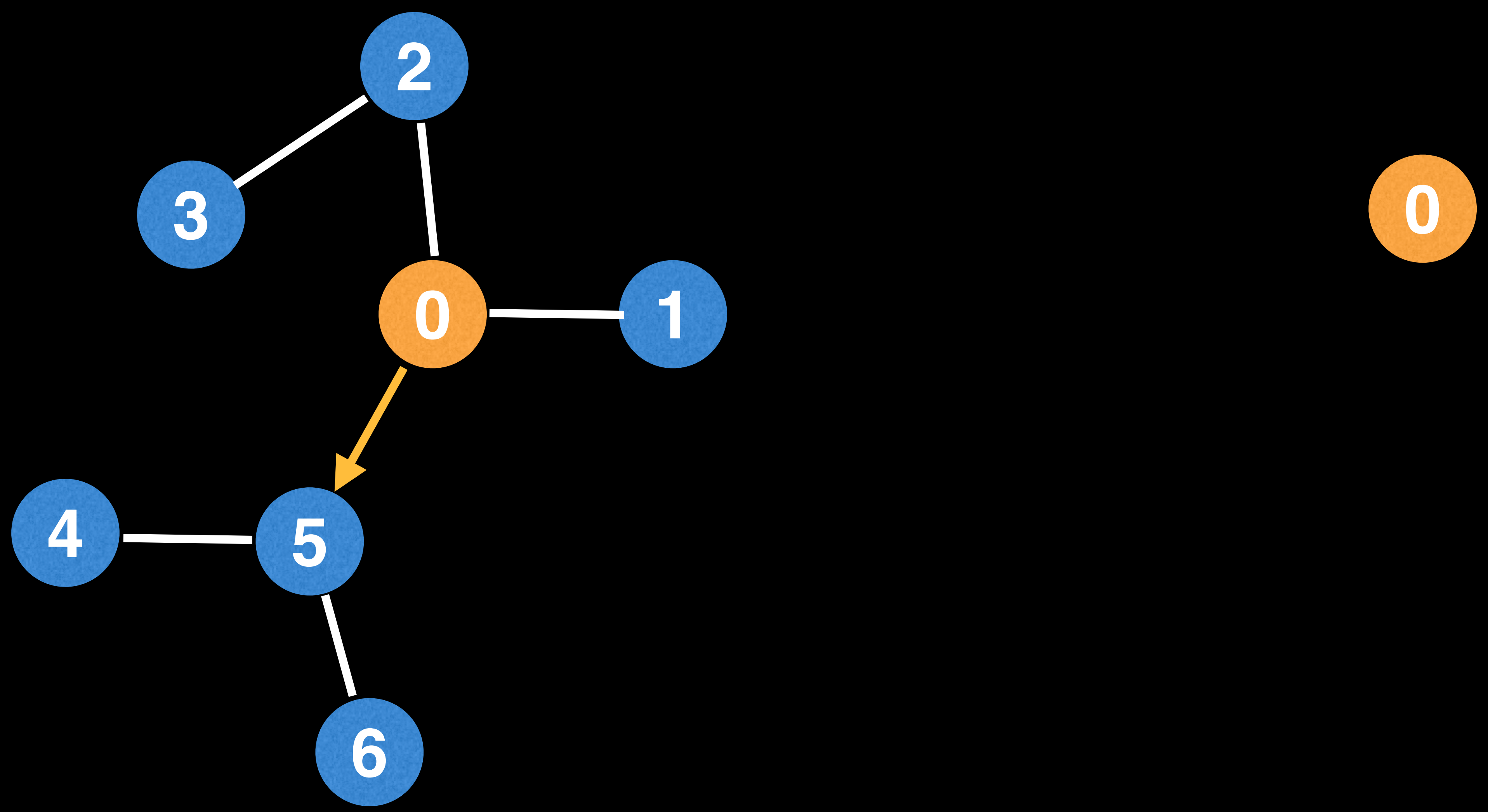
Rooting a tree is easily done depth first.



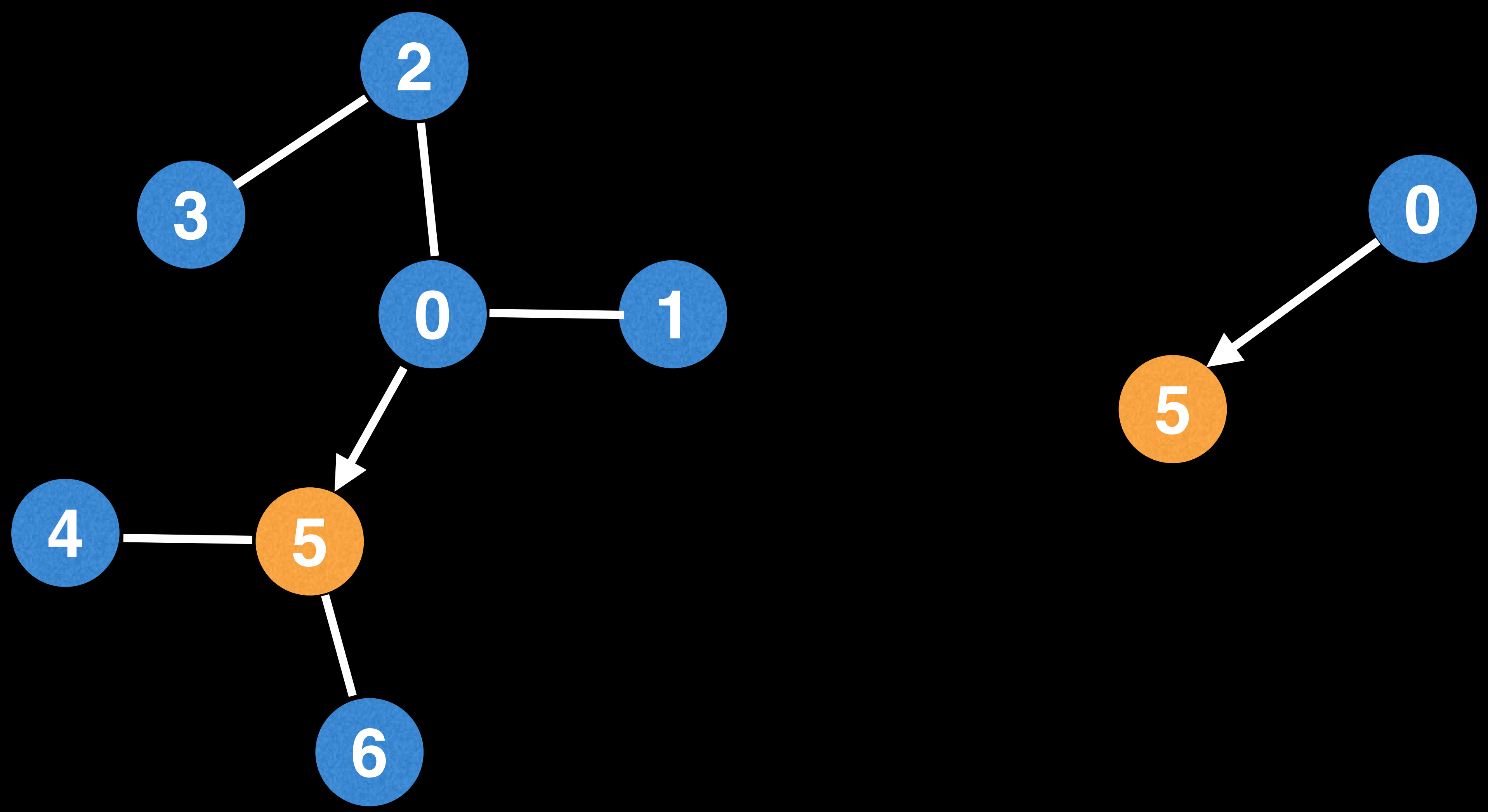
Rooting a tree is easily done depth first.



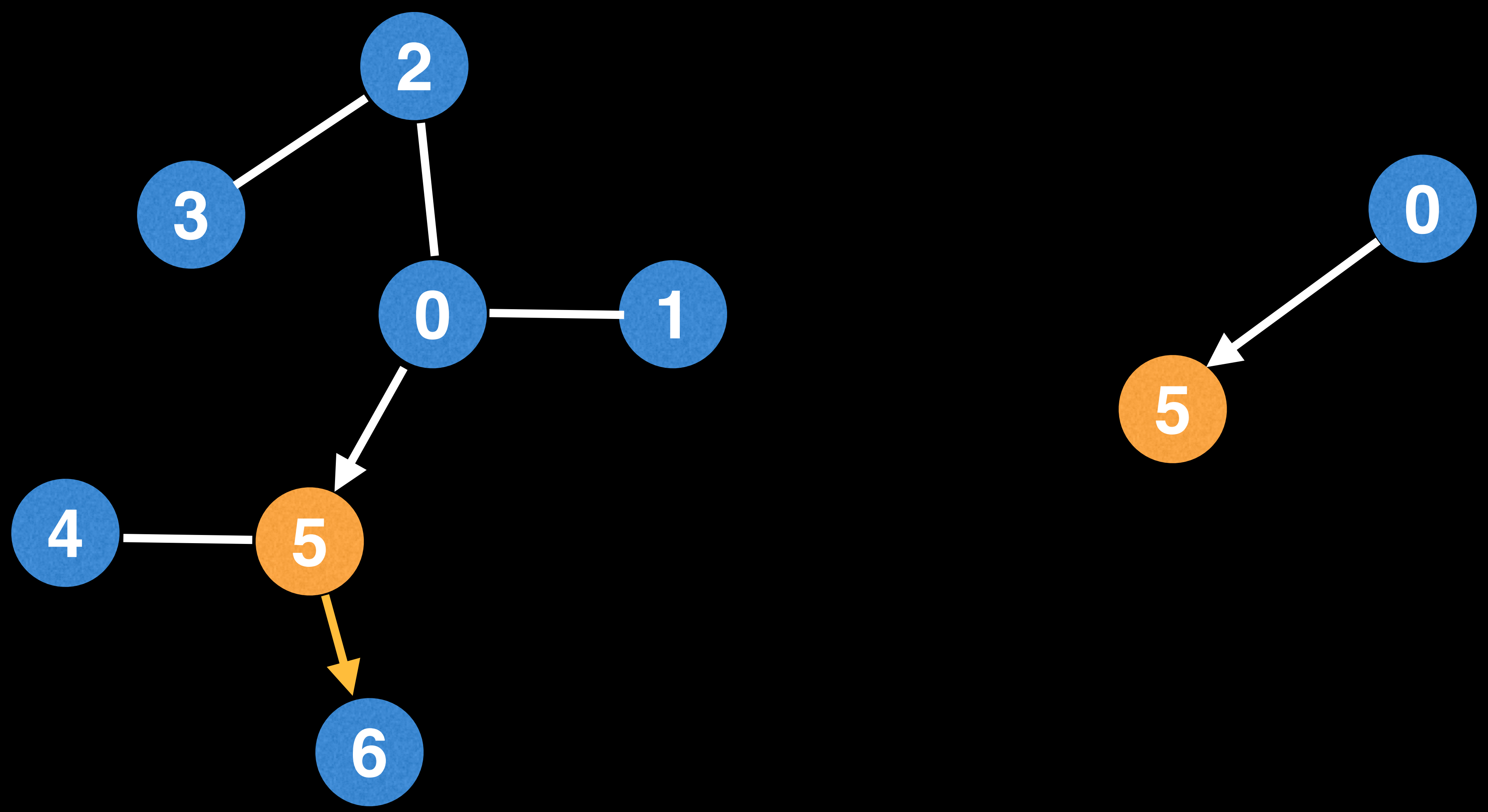
Rooting a tree is easily done depth first.



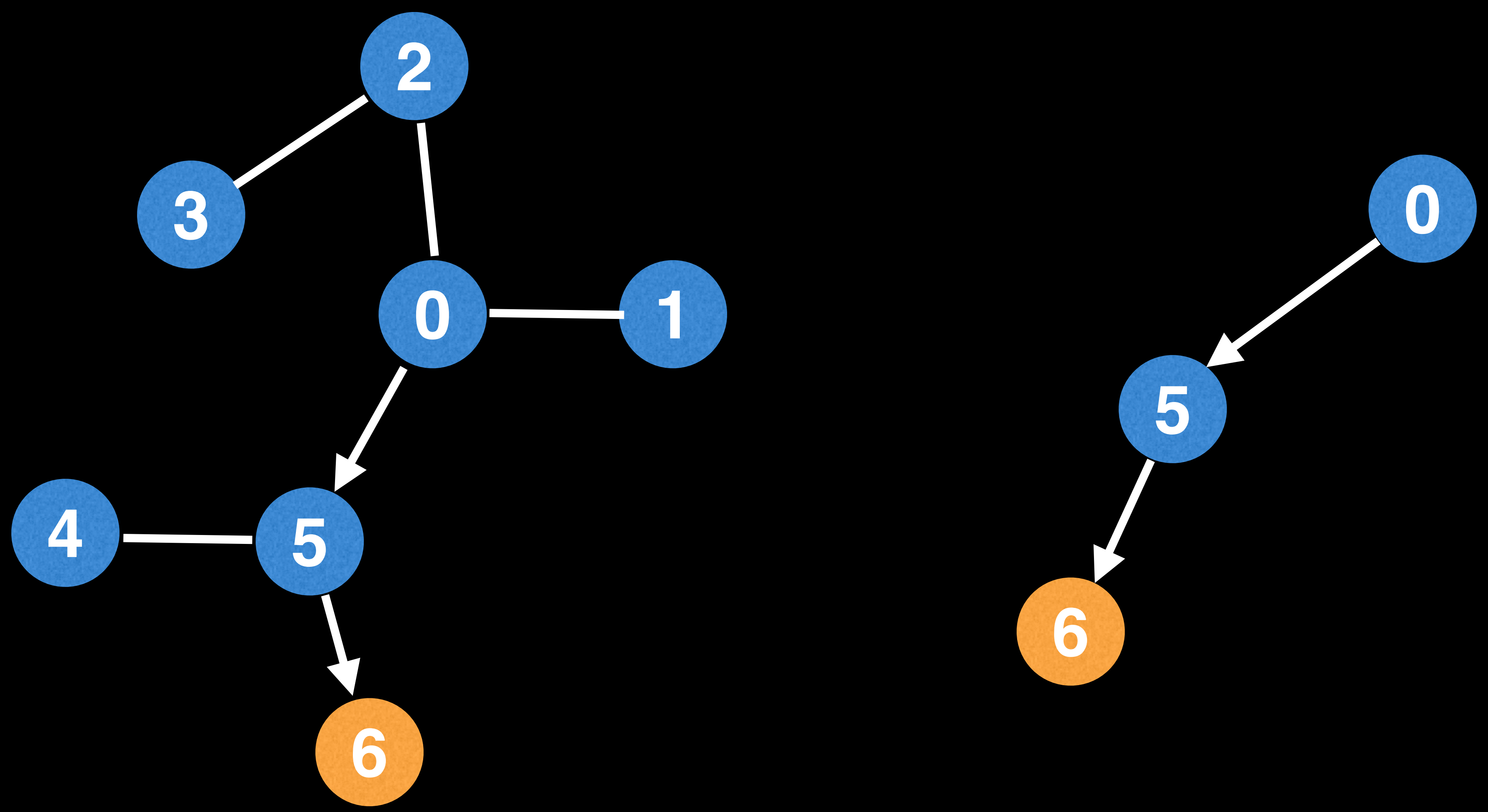
Rooting a tree is easily done depth first.



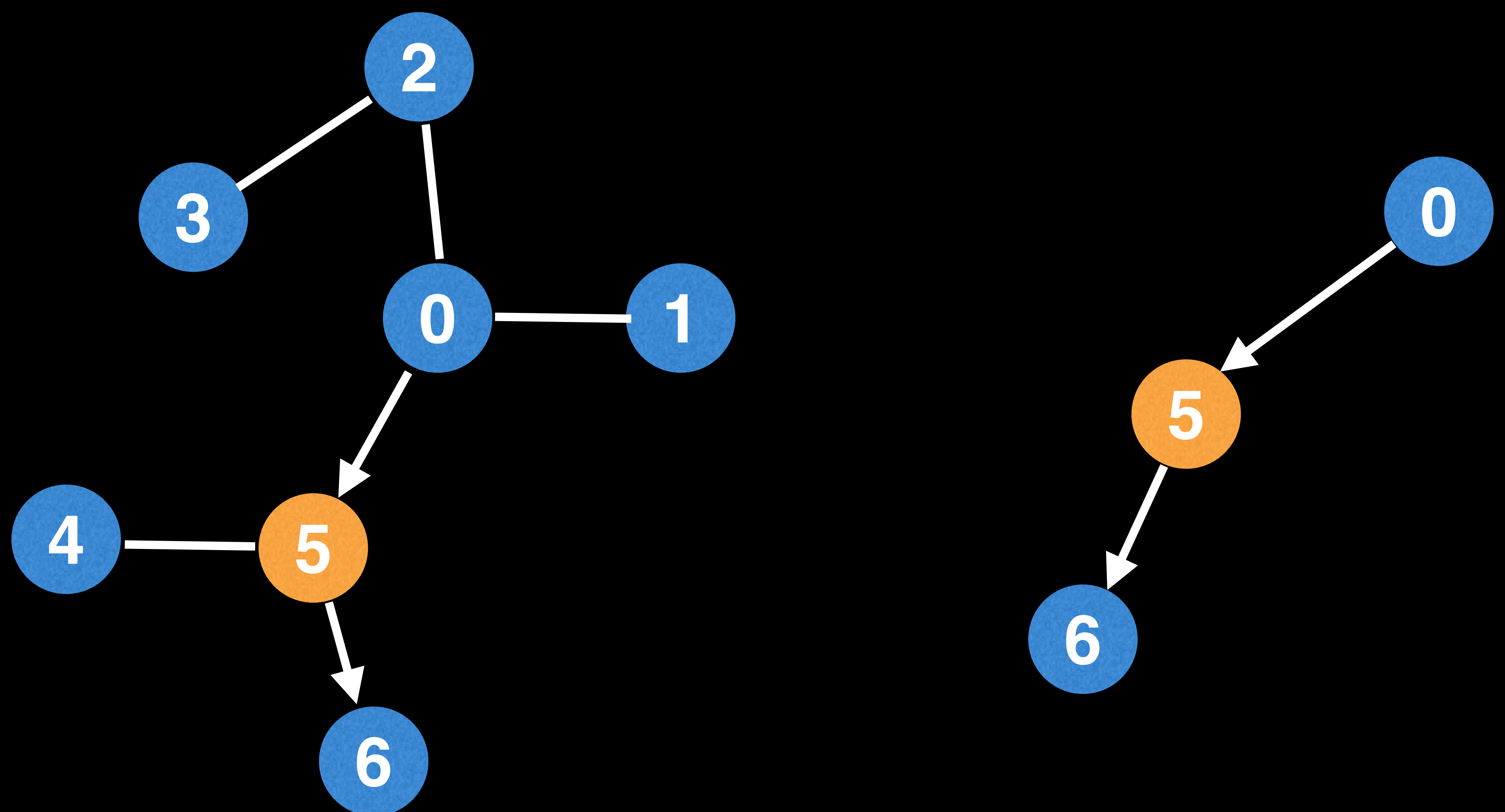
Rooting a tree is easily done depth first.



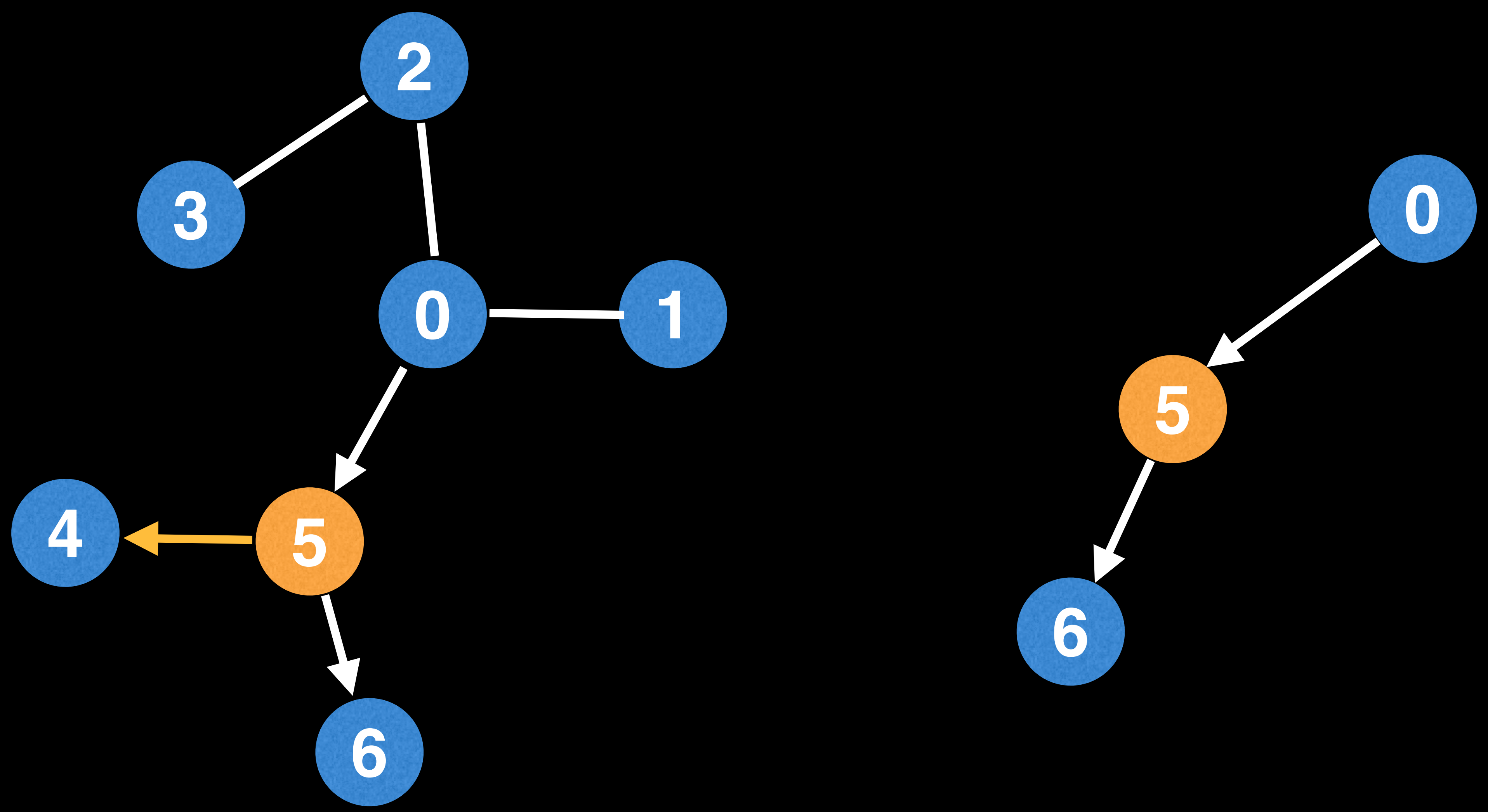
Rooting a tree is easily done depth first.



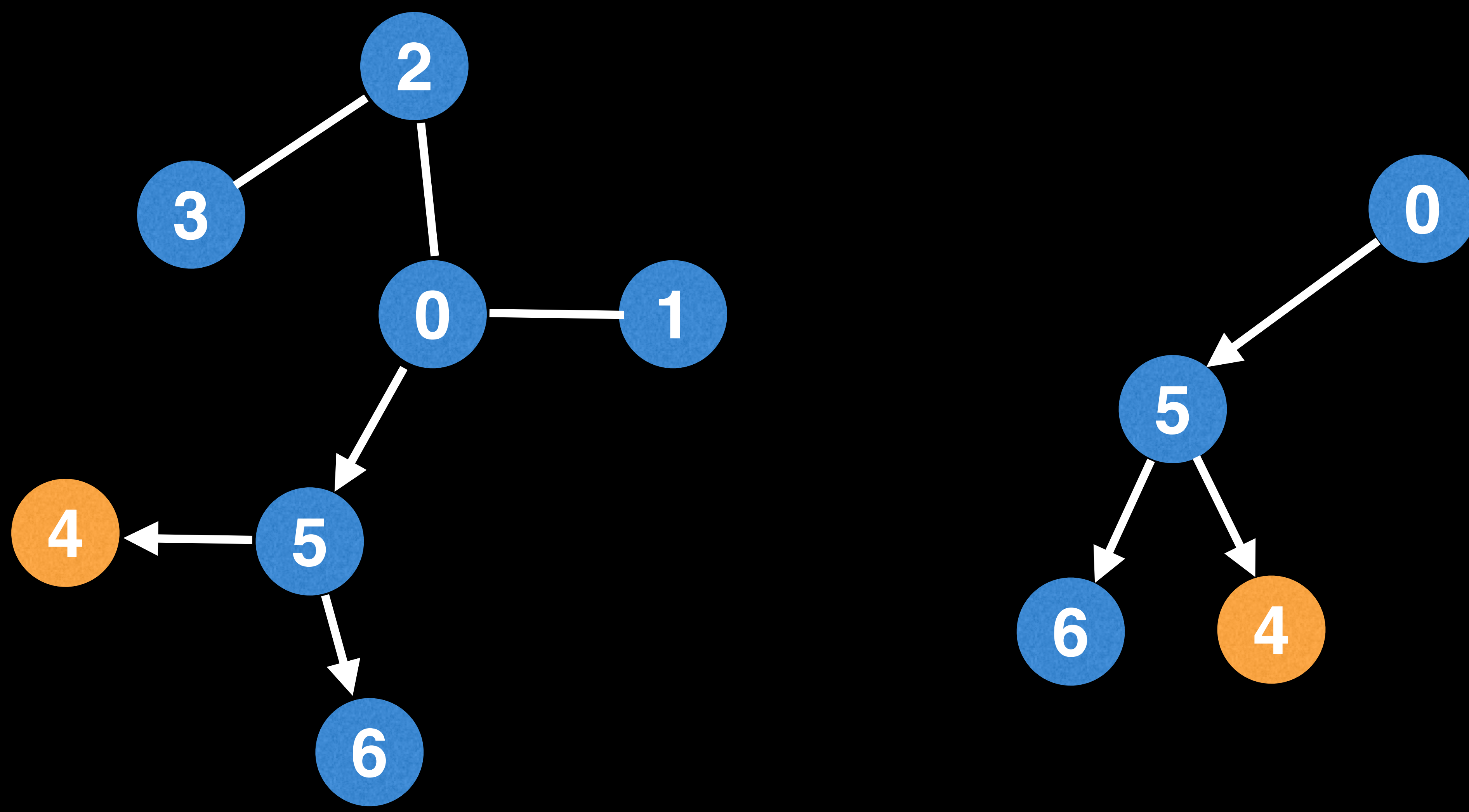
Rooting a tree is easily done depth first.



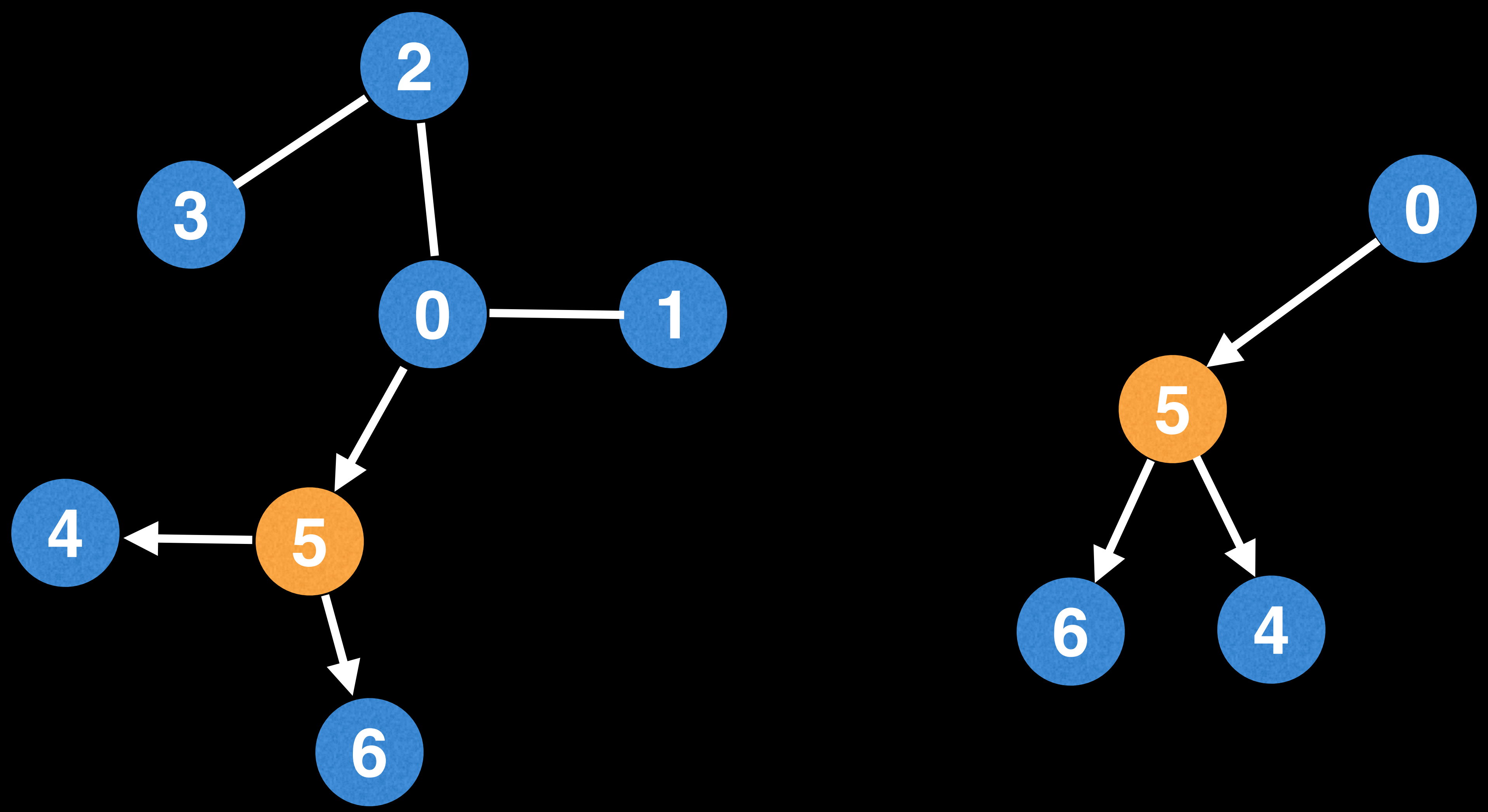
Rooting a tree is easily done depth first.



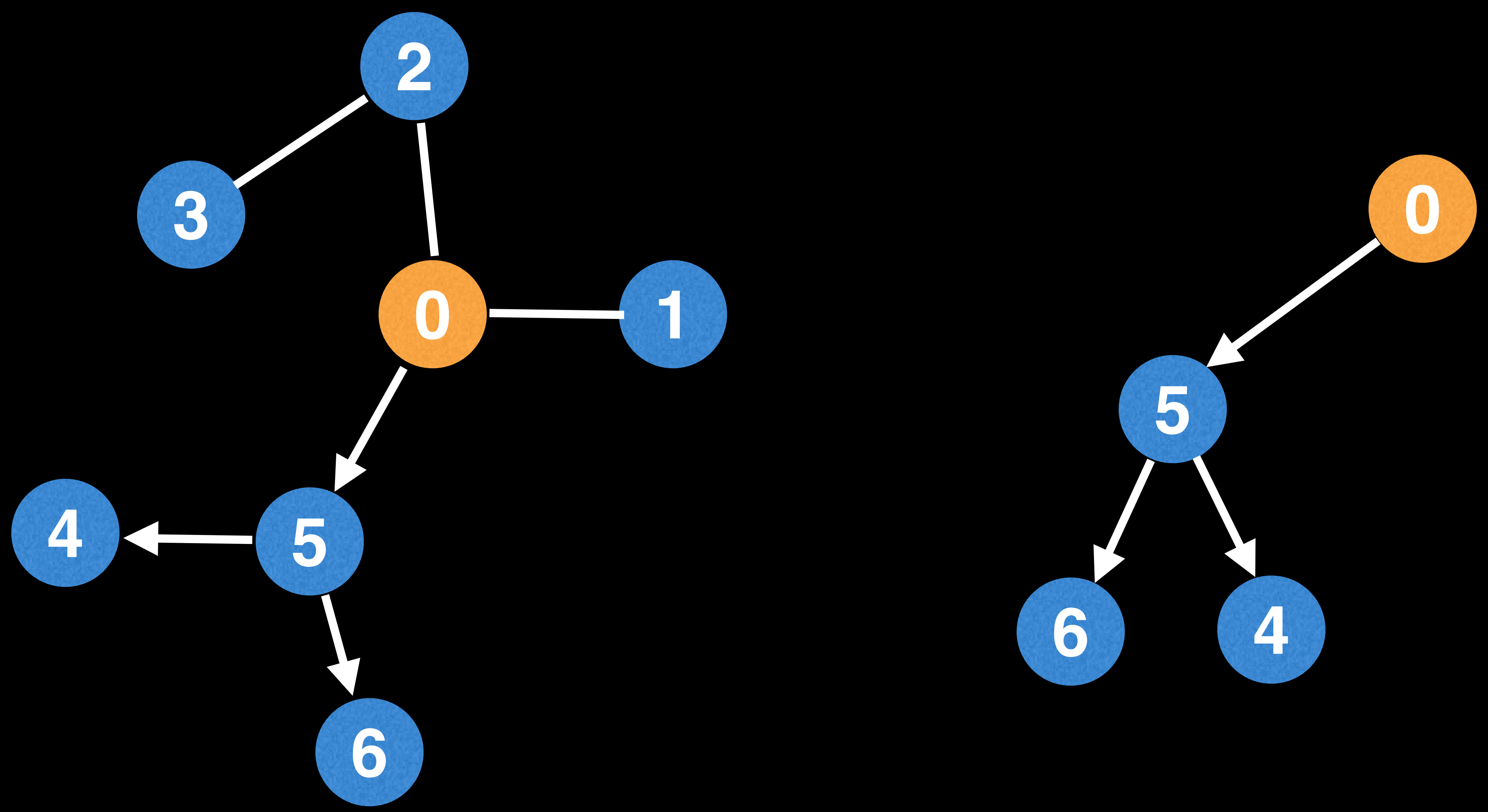
Rooting a tree is easily done depth first.



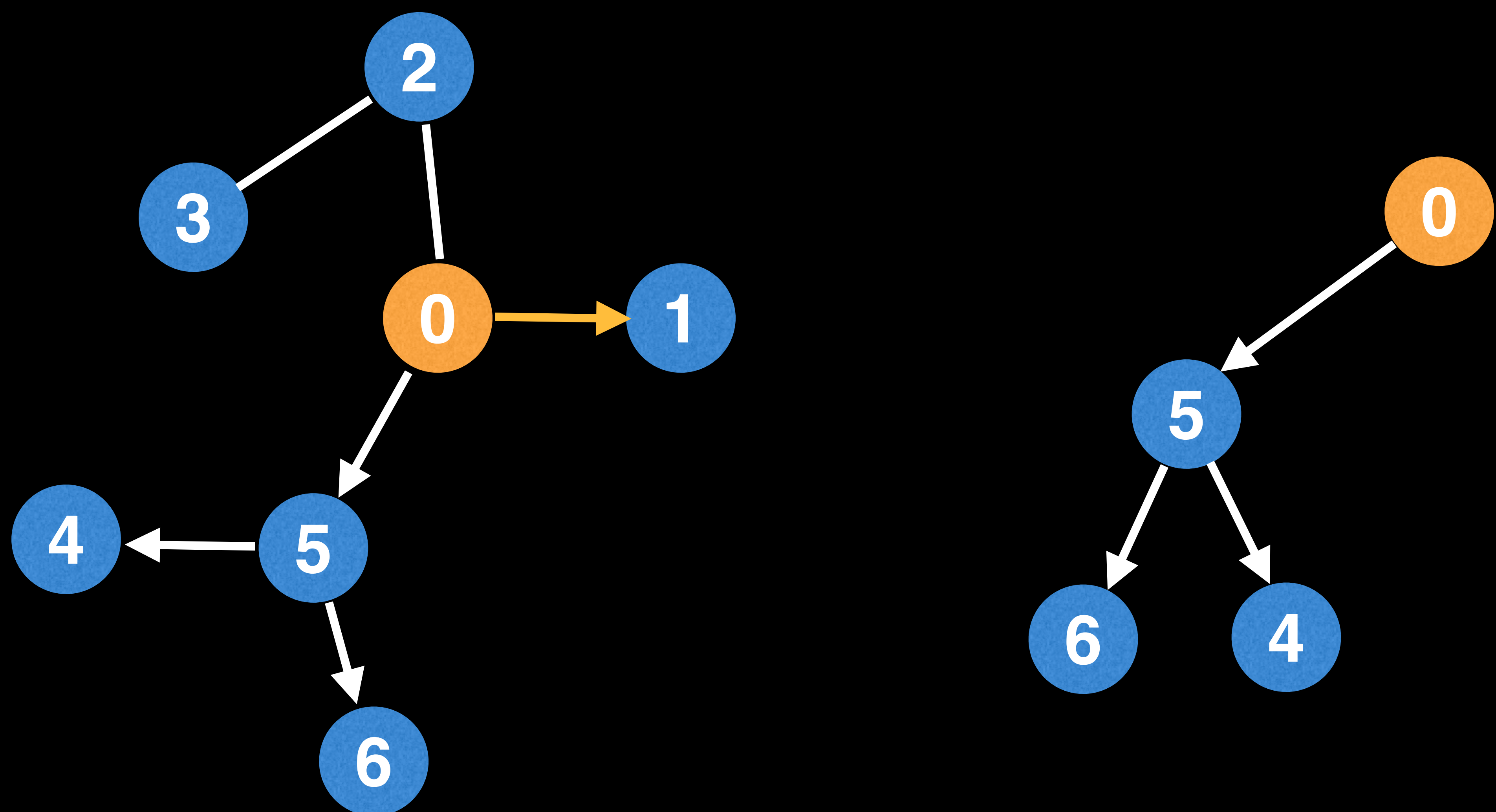
Rooting a tree is easily done depth first.



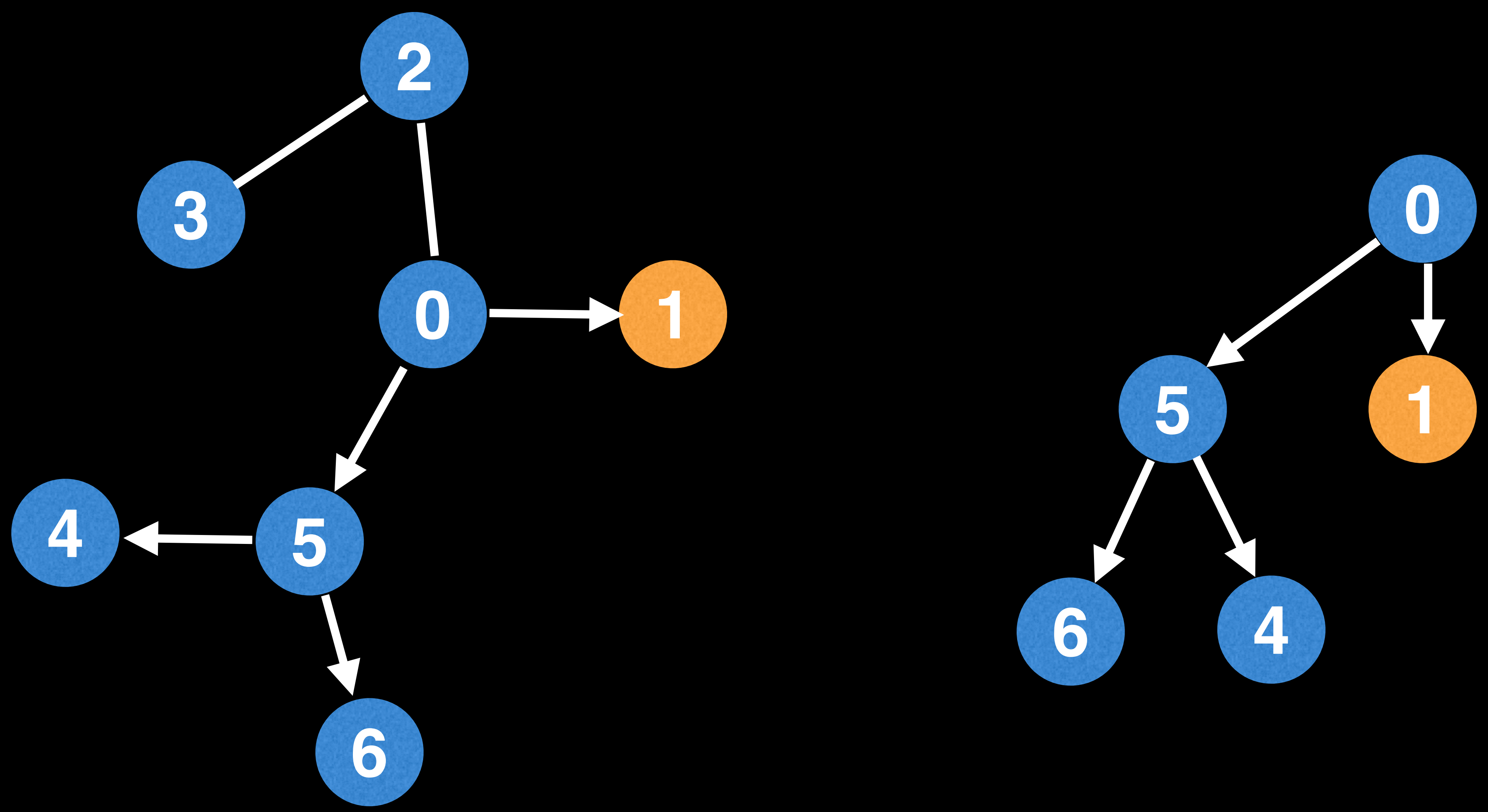
Rooting a tree is easily done depth first.



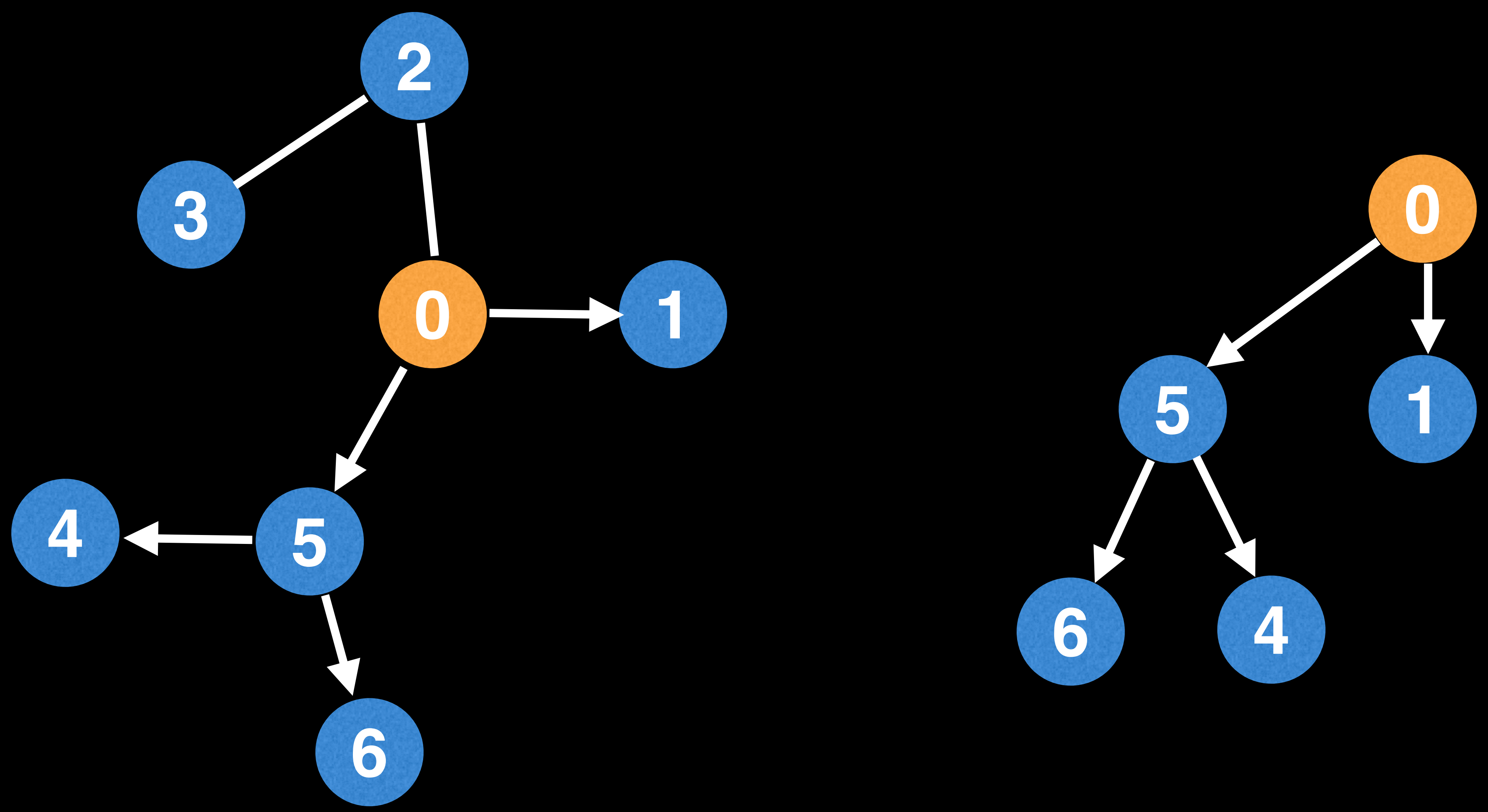
Rooting a tree is easily done depth first.



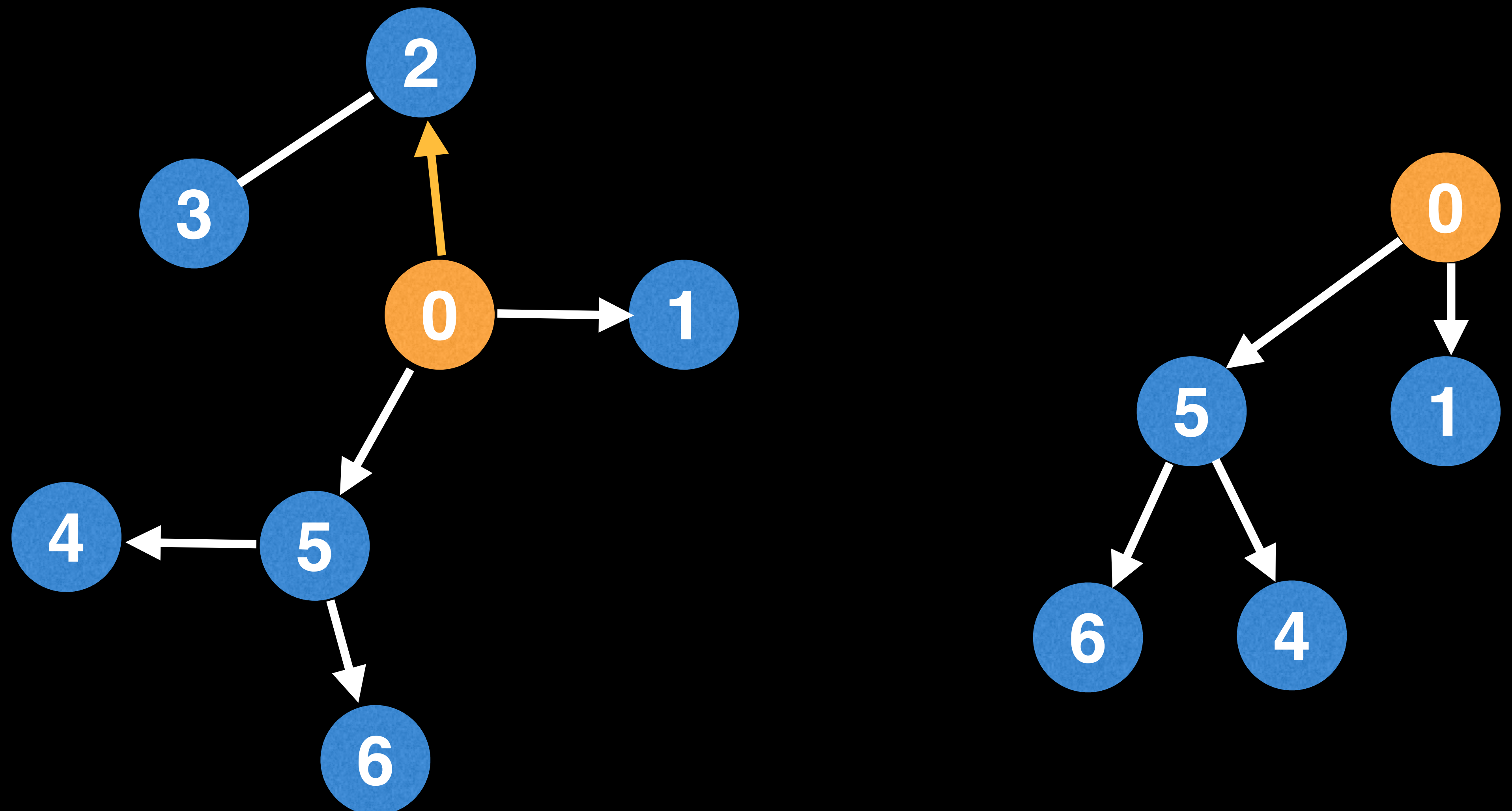
Rooting a tree is easily done depth first.



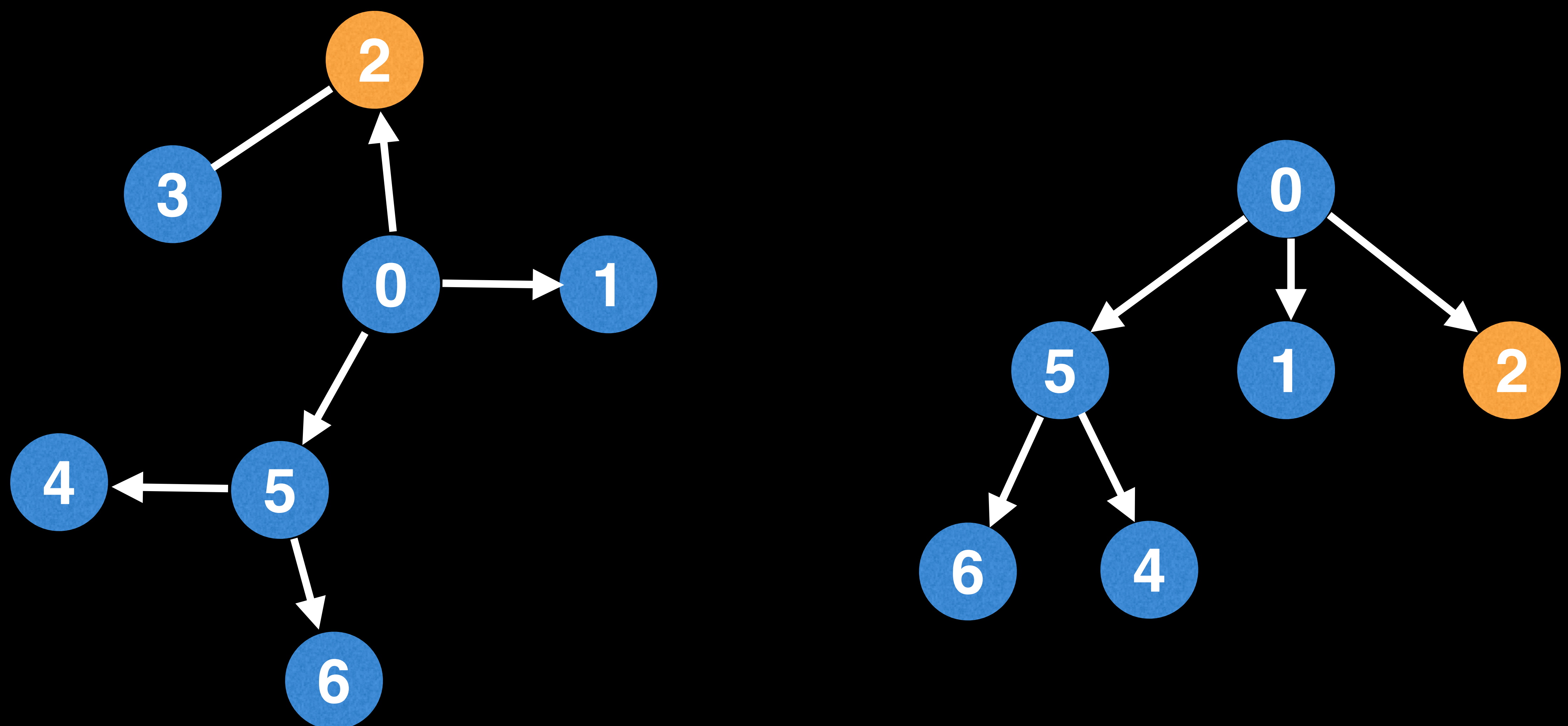
Rooting a tree is easily done depth first.



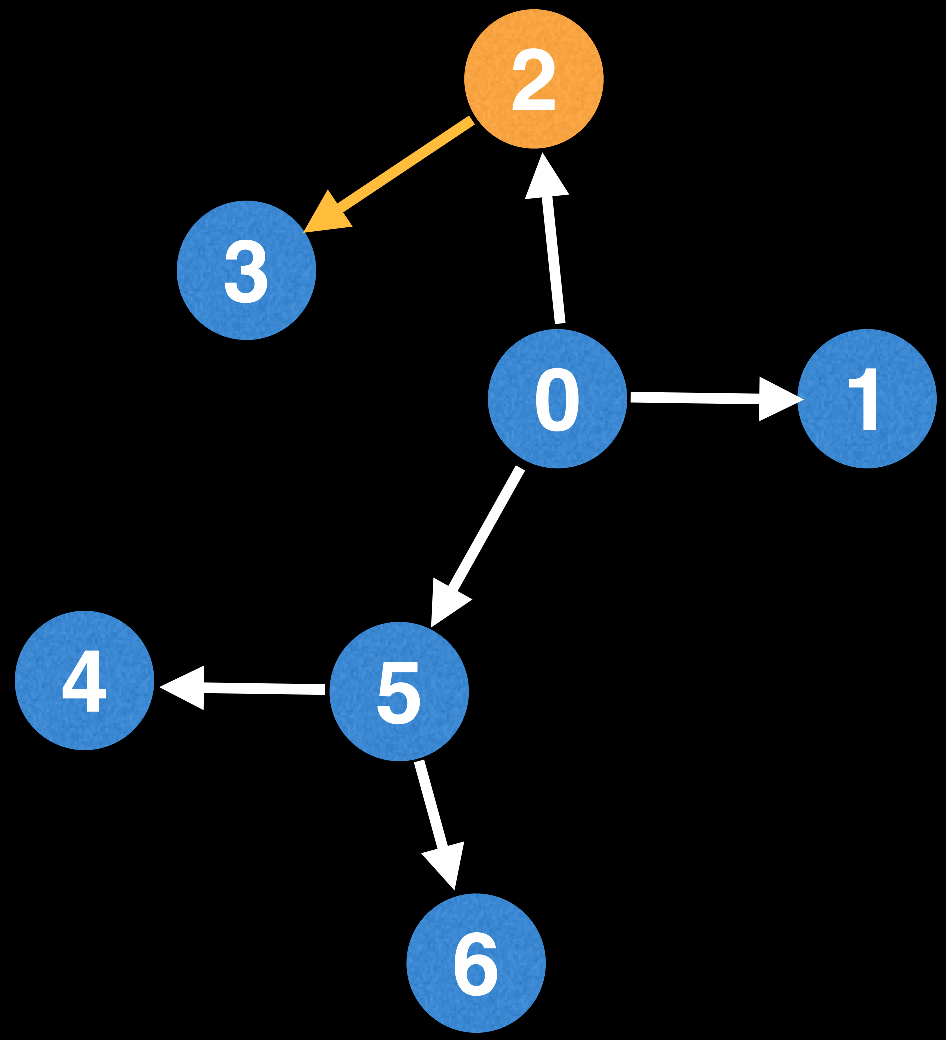
Rooting a tree is easily done depth first.

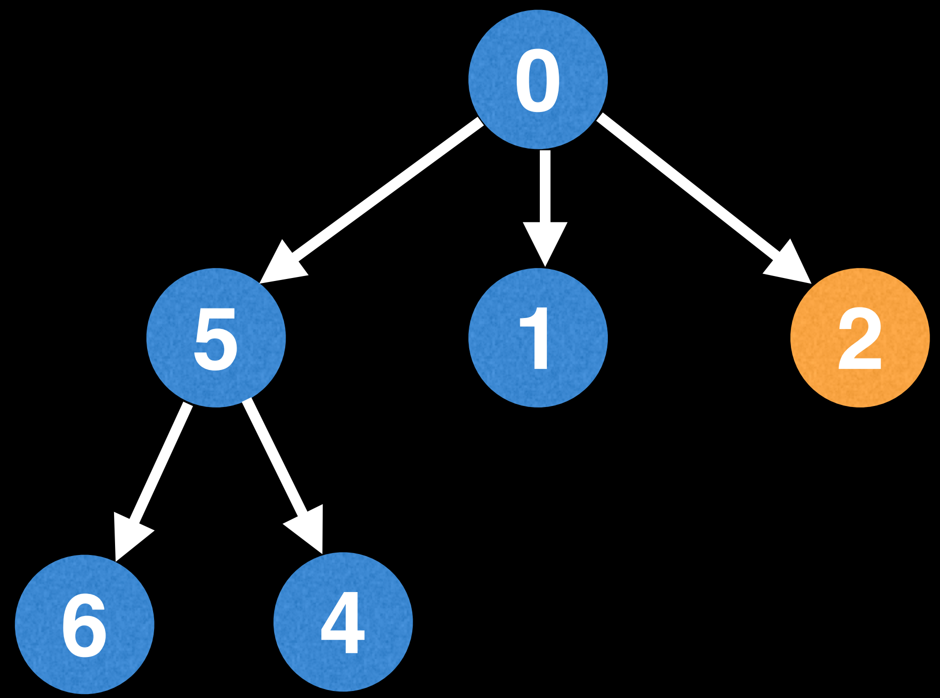


Rooting a tree is easily done depth first.

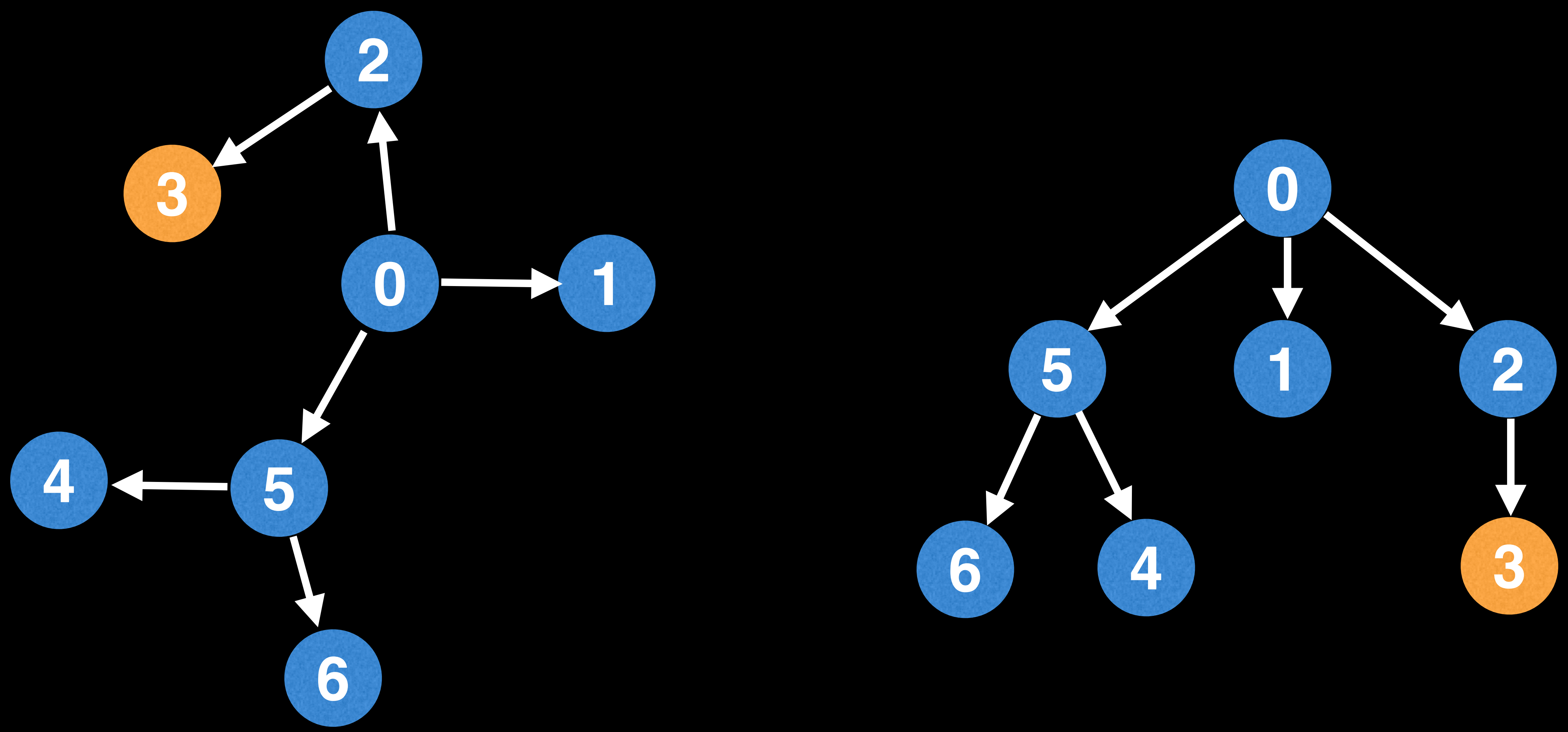


Rooting a tree is easily done depth first.

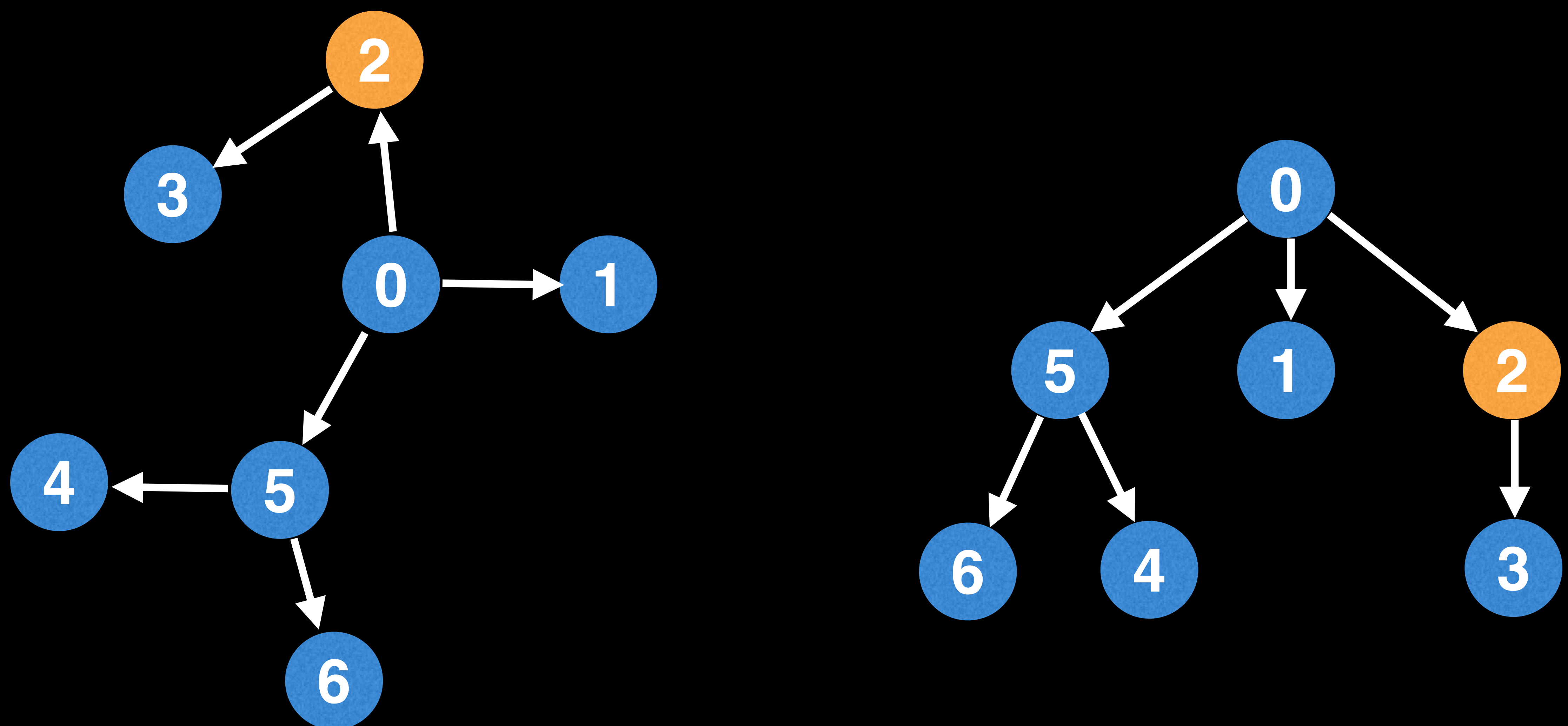




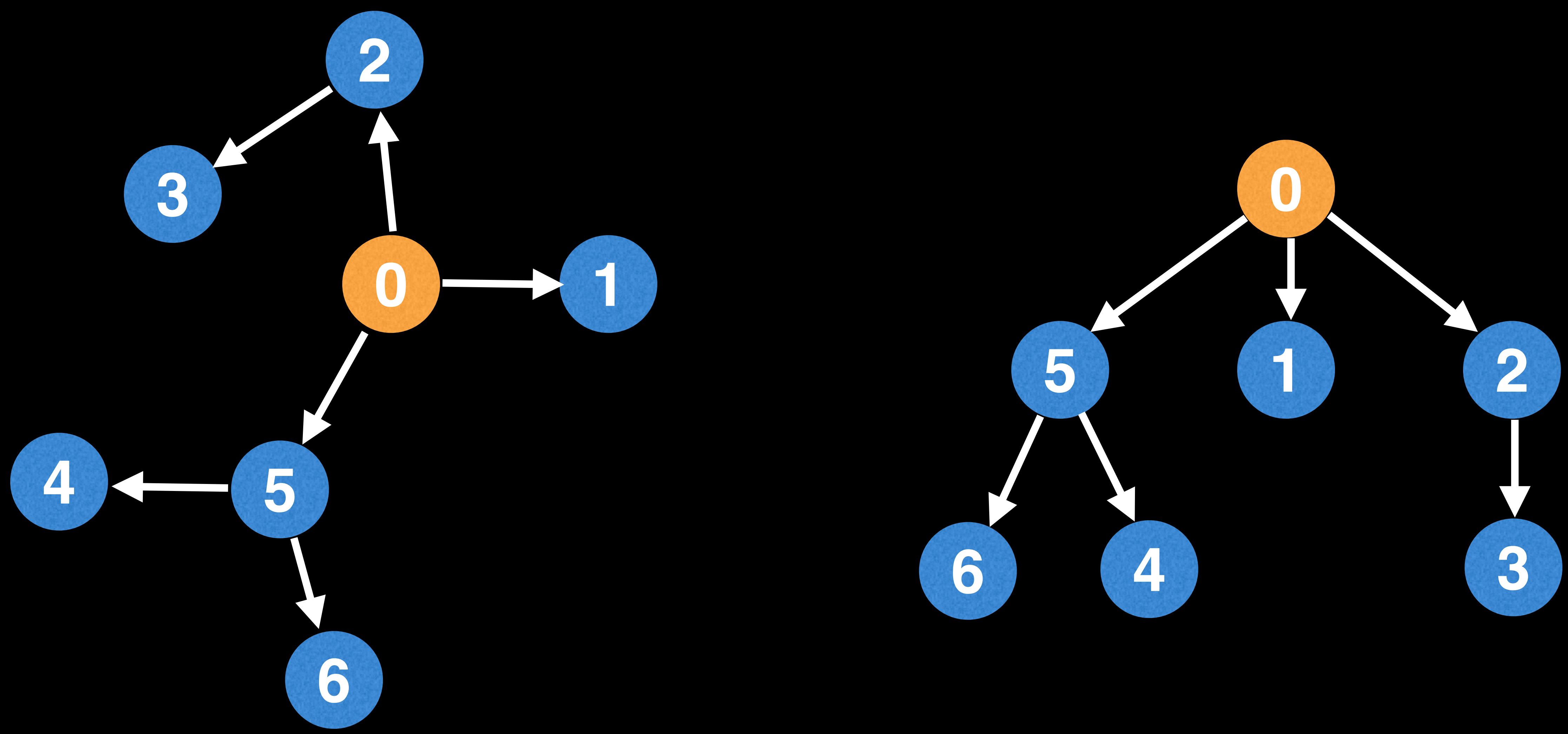
Rooting a tree is easily done depth first.



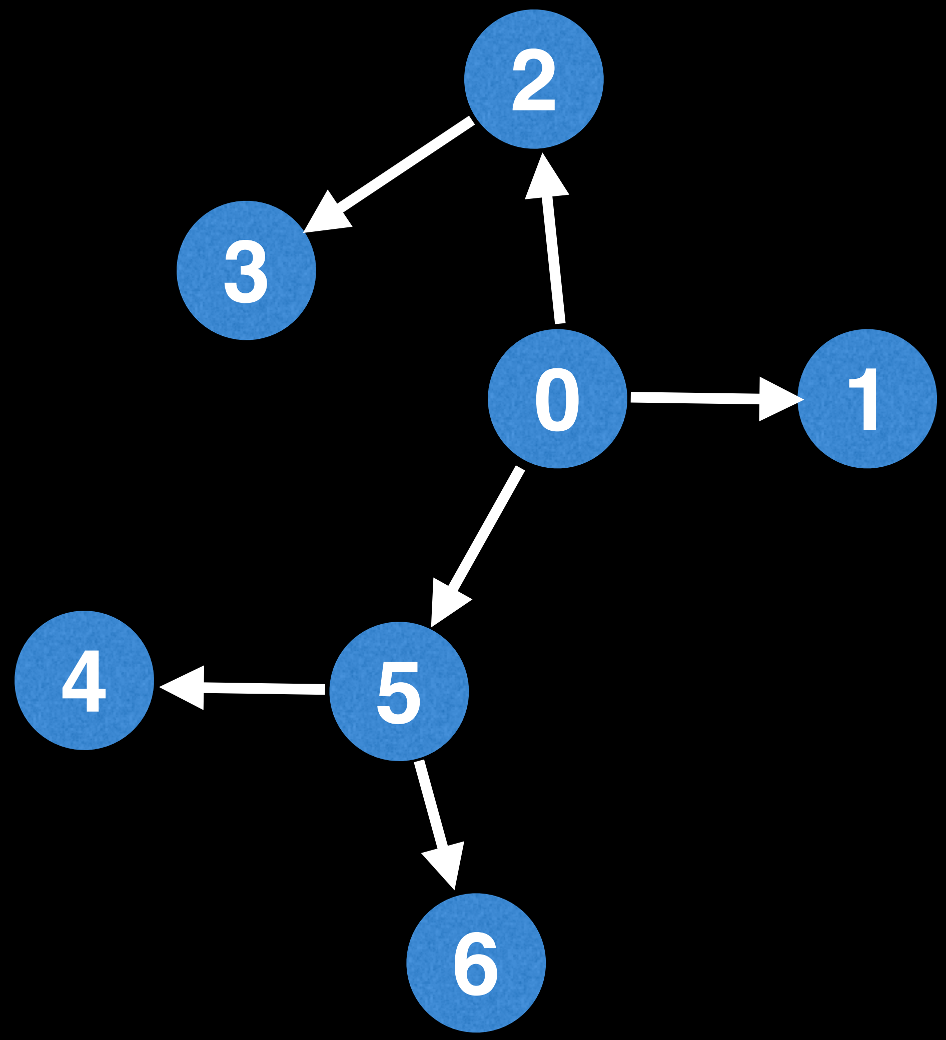
Rooting a tree is easily done depth first.

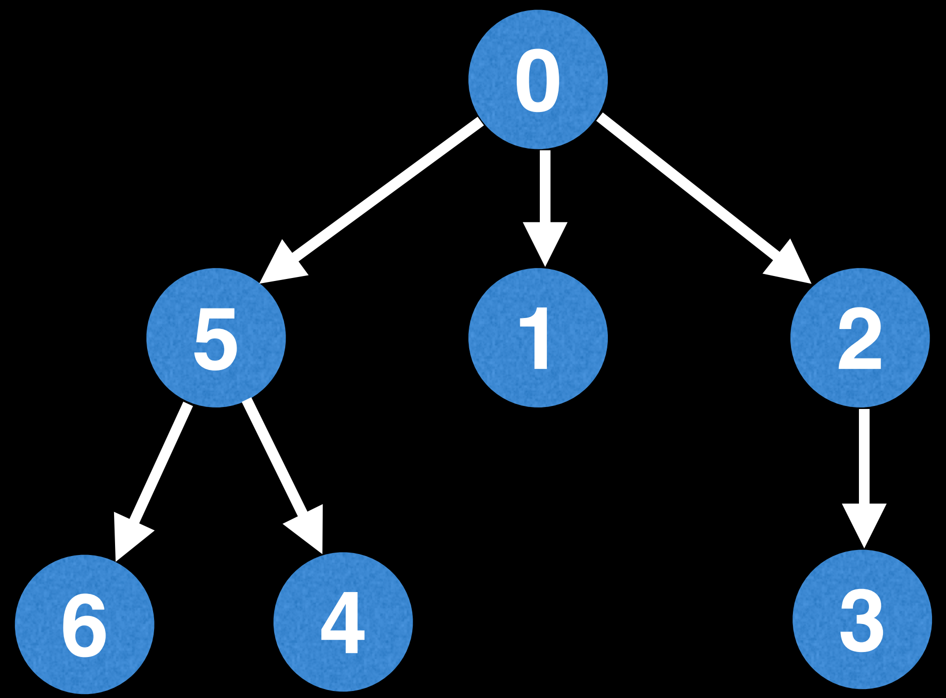


Rooting a tree is easily done depth first.



Rooting a tree is easily done depth first.





Rooting tree pseudocode

```
# TreeNode object structure.
class TreeNode:
  # Unique integer id to identify this node.
  int id;

  # Pointer to parent TreeNode reference. Only the
  # root node has a null parent TreeNode reference.
  TreeNode parent;

  # List of pointers to child TreeNodes.
  TreeNode[] children;
```

Rooting tree pseudocode

```
# TreeNode object structure.
class TreeNode:
  # Unique integer id to identify this node.
  int id;

  # Pointer to parent TreeNode reference. Only the
  # root node has a null parent TreeNode reference.
  TreeNode parent;

  # List of pointers to child TreeNodes.
  TreeNode[] children;
```

Rooting tree pseudocode

```
# TreeNode object structure.
class TreeNode:
  # Unique integer id to identify this node.
  int id;

  # Pointer to parent TreeNode reference. Only the
  # root node has a null parent TreeNode reference.
  TreeNode parent;

  # List of pointers to child TreeNodes.
  TreeNode[] children;
```

Rooting tree pseudocode

```
# TreeNode object structure.
class TreeNode:
    # Unique integer id to identify this node.
    int id;

    # Pointer to parent TreeNode reference. Only the
    # root node has a null parent TreeNode reference.
    TreeNode parent;

    # List of pointers to child TreeNodes.
    TreeNode[] children;
```

Rooting tree pseudocode

```
# TreeNode object structure.
class TreeNode:
  # Unique integer id to identify this node.
  int id;

  # Pointer to parent TreeNode reference. Only the
  # root node has a null parent TreeNode reference.
  TreeNode parent;

  # List of pointers to child TreeNodes.
  TreeNode[] children;
```

Rooting tree pseudocode

```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there’s an edge between
# (u, v) there’s also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

Rooting tree pseudocode

```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there’s an edge between
# (u, v) there’s also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

Rooting tree pseudocode

```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there’s an edge between
# (u, v) there’s also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

Rooting tree pseudocode

```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there’s an edge between
# (u, v) there’s also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

Rooting tree pseudocode

```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there’s an edge between
# (u, v) there’s also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

Rooting tree pseudocode

```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there’s an edge between
# (u, v) there’s also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

Rooting tree pseudocode

```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there's an edge between
# (u, v) there's also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

root node has
no parent!

Rooting tree pseudocode

```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there’s an edge between
# (u, v) there’s also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

Rooting tree pseudocode

```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there’s an edge between
# (u, v) there’s also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

Rooting tree pseudocode

```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there's an edge between
# (u, v) there's also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

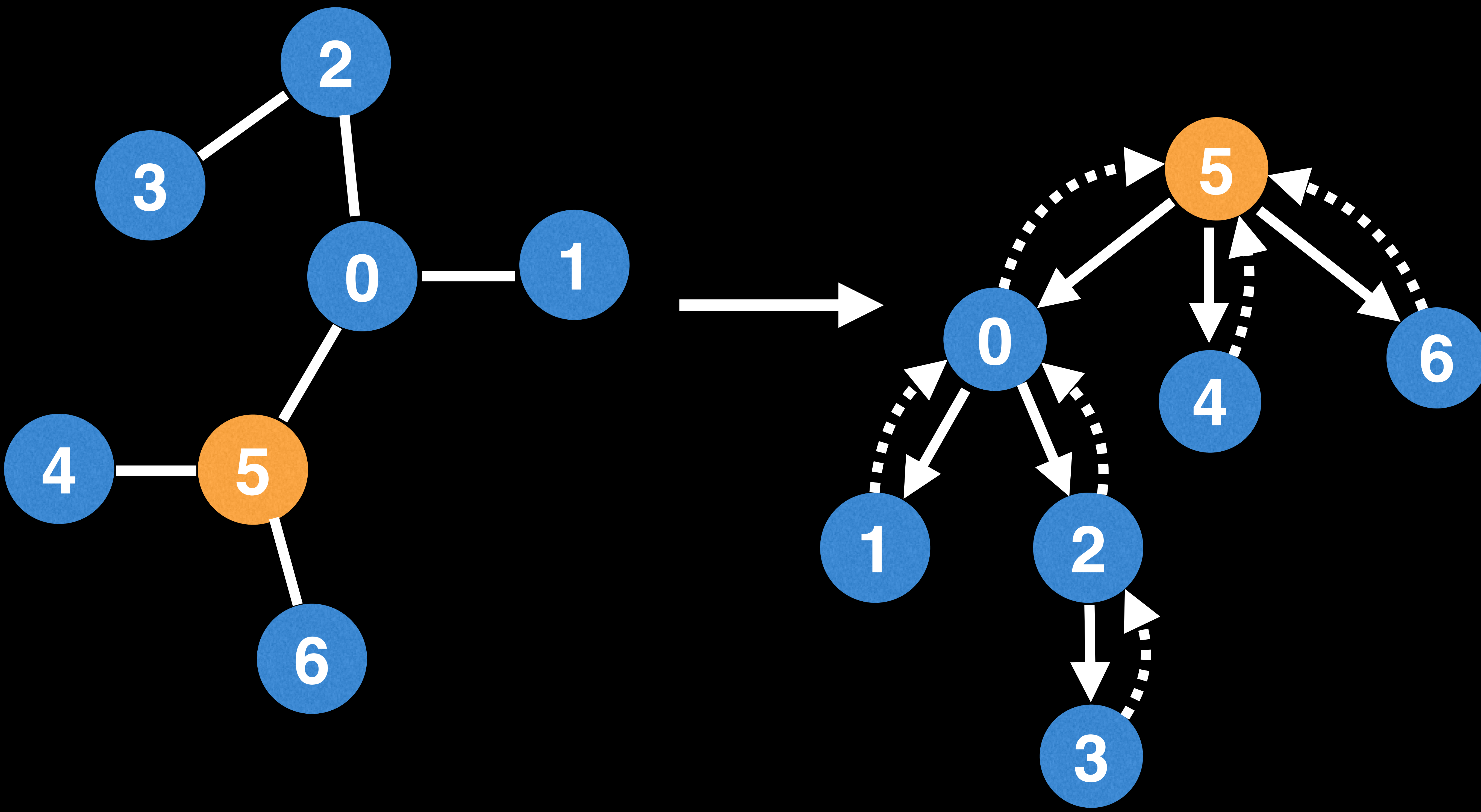
Rooting tree pseudocode

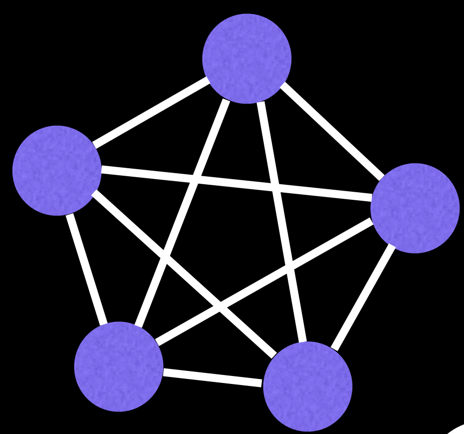
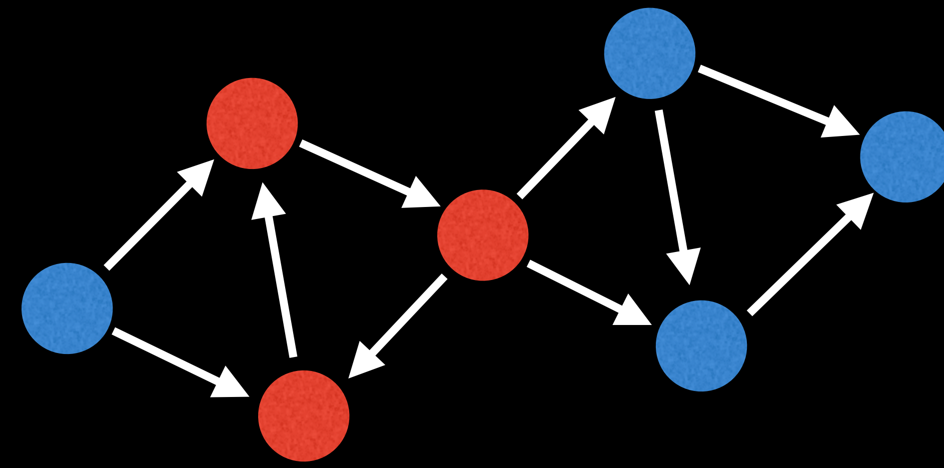
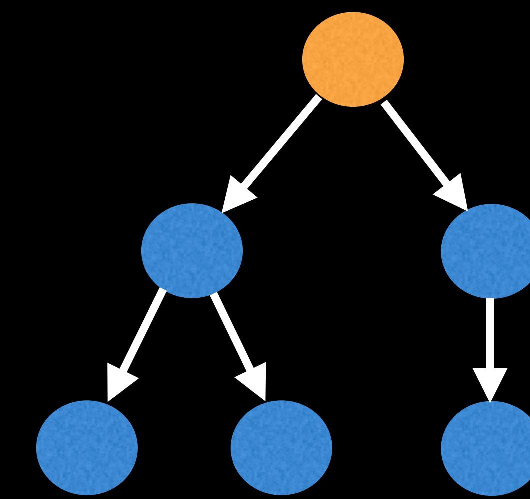
```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there’s an edge between
# (u, v) there’s also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

Rooting tree pseudocode

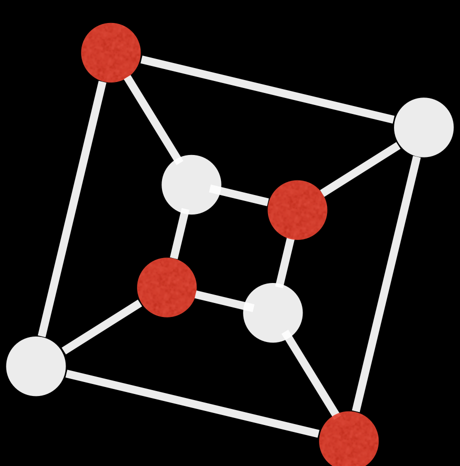
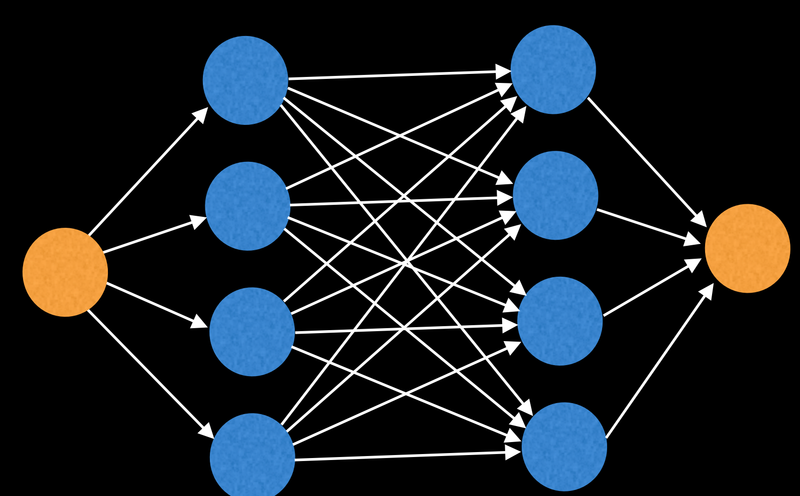
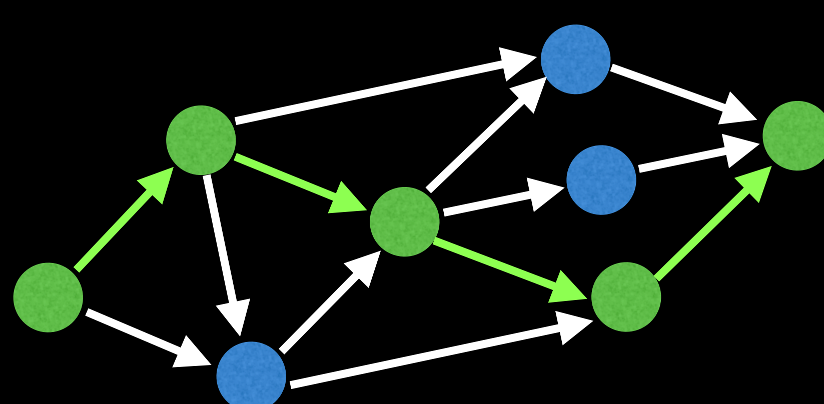
```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there’s an edge between
# (u, v) there’s also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

Rooting a Tree



Graph Theory Video Series

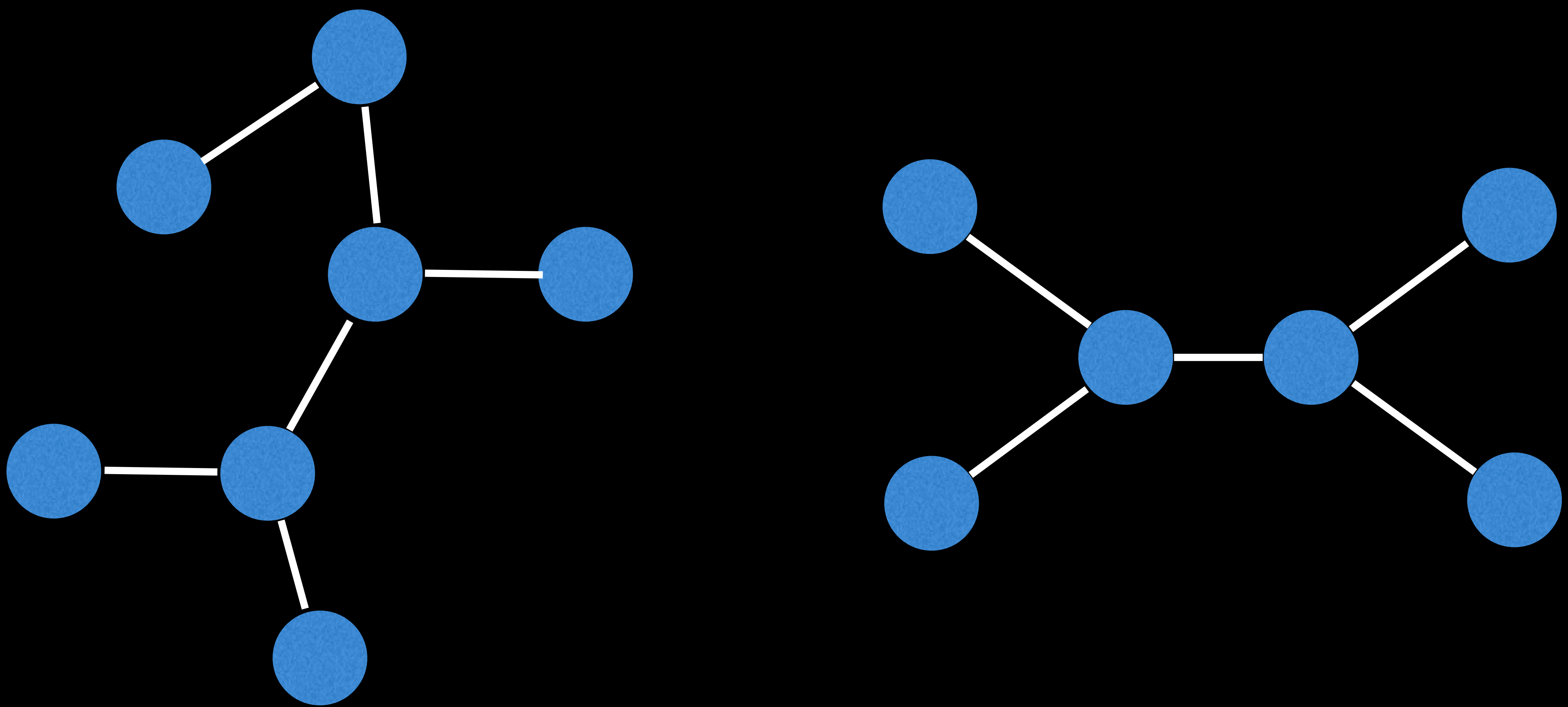




Center(s) of a tree

William Fiset

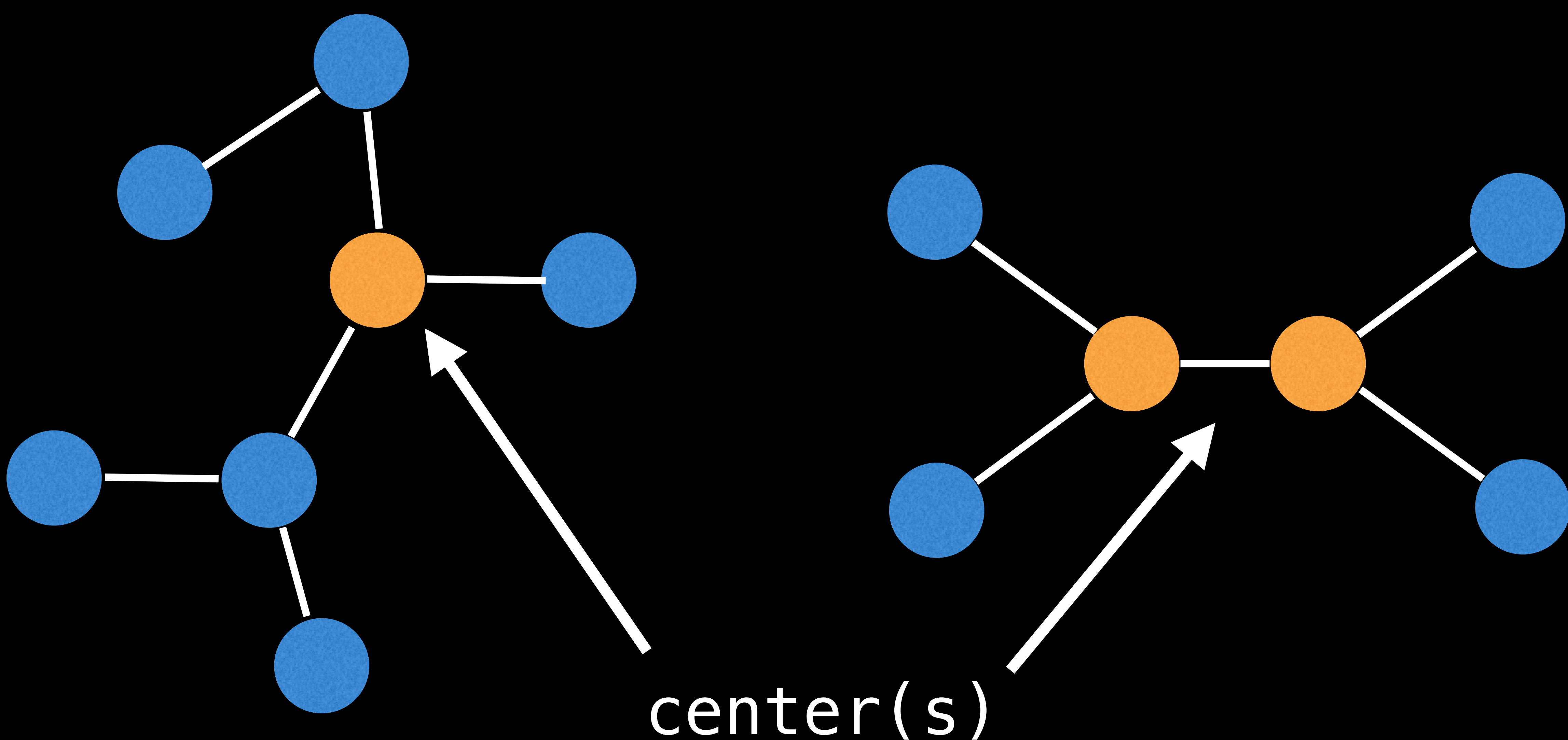
Center(s) of undirected tree

An interesting problem when you have an undirected tree is finding the tree’s **center node(s)**. This could come in handy if we wanted to select a good node to root our tree 😉

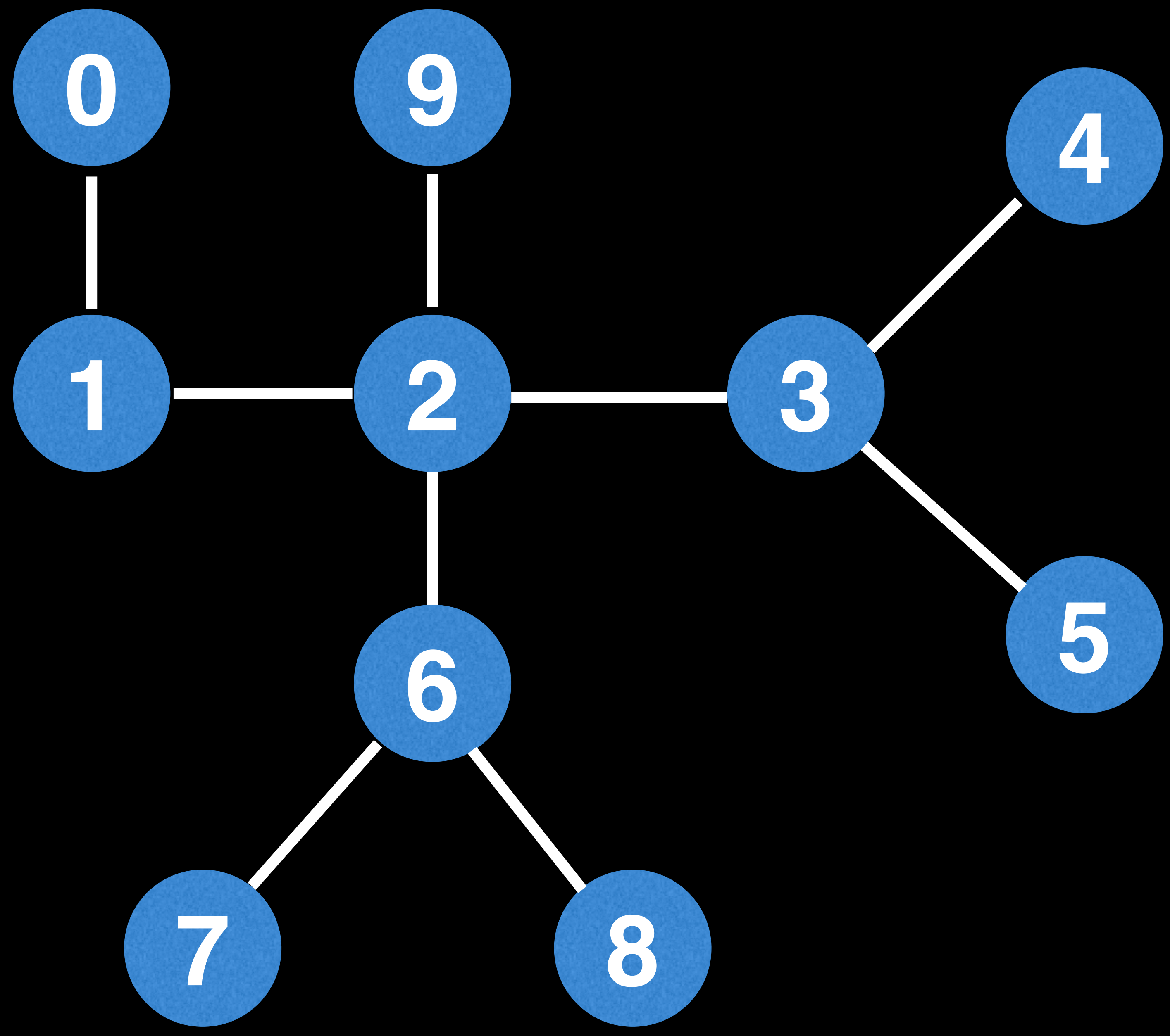


Center(s) of undirected tree

An interesting problem when you have an undirected tree is finding the tree's **center node(s)**. This could come in handy if we wanted to select a good node to root our tree 😉

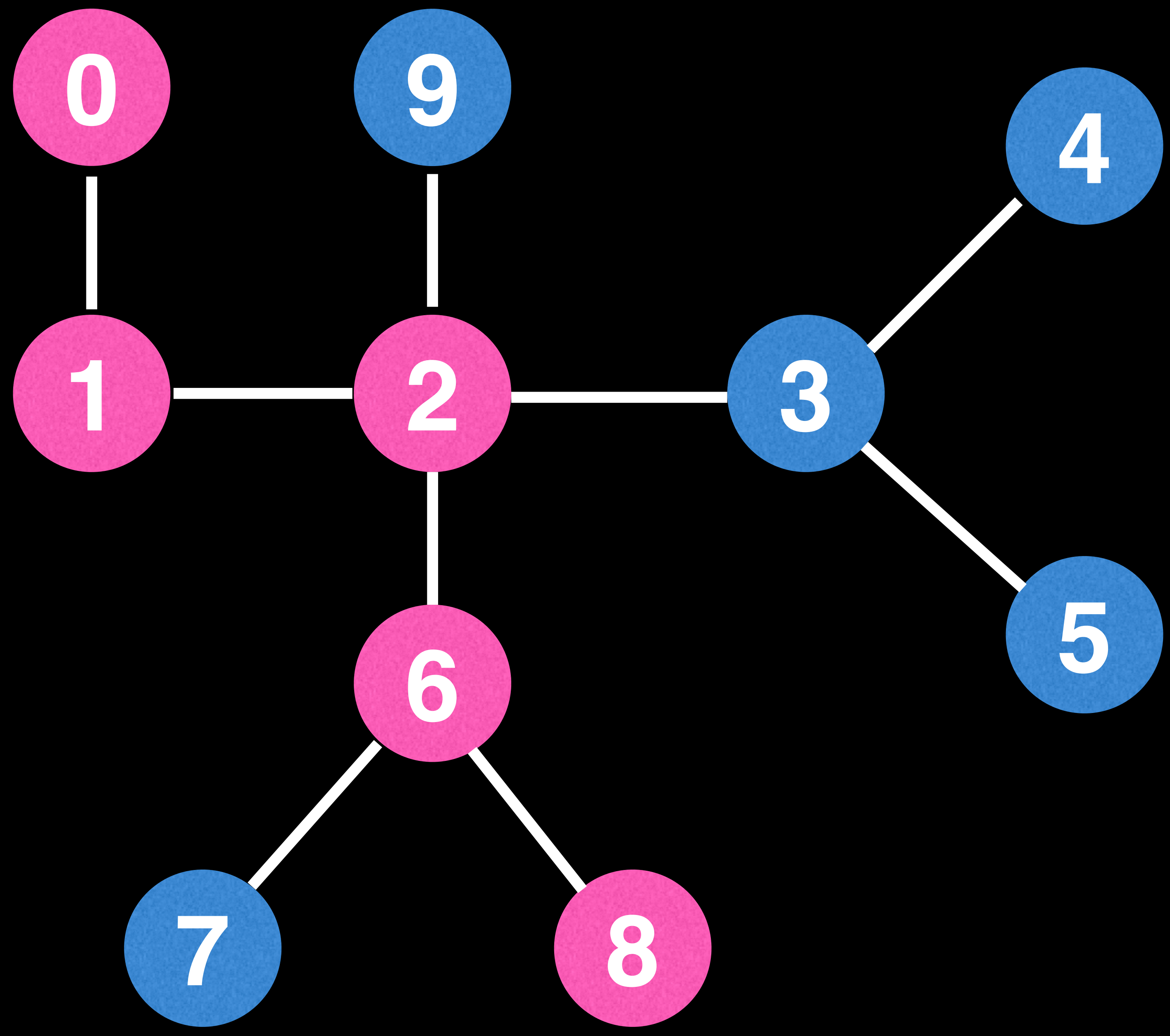


Center(s) of undirected tree



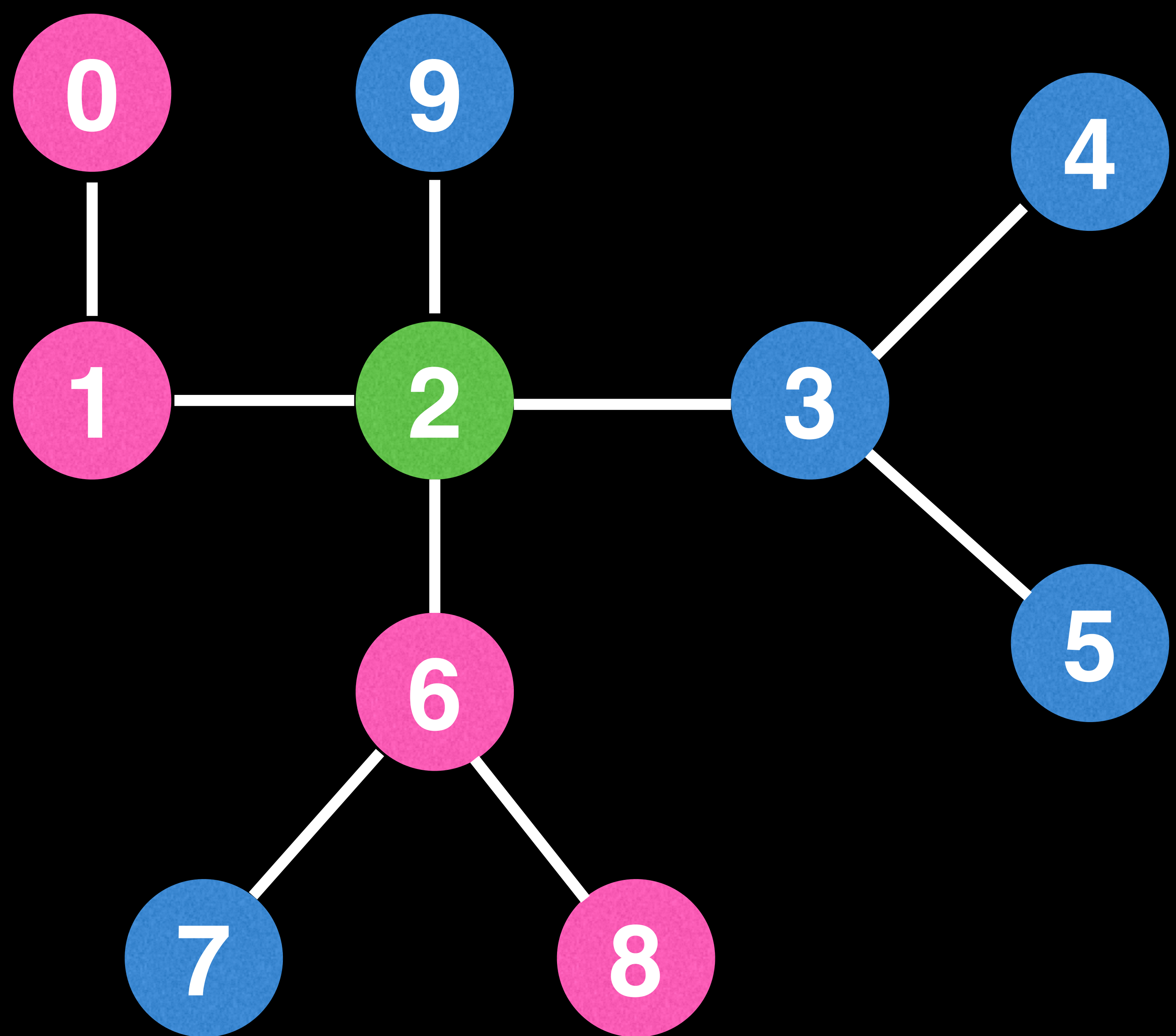
Notice that the center is always the middle vertex or middle two vertices in every longest path along the tree.

Center(s) of undirected tree



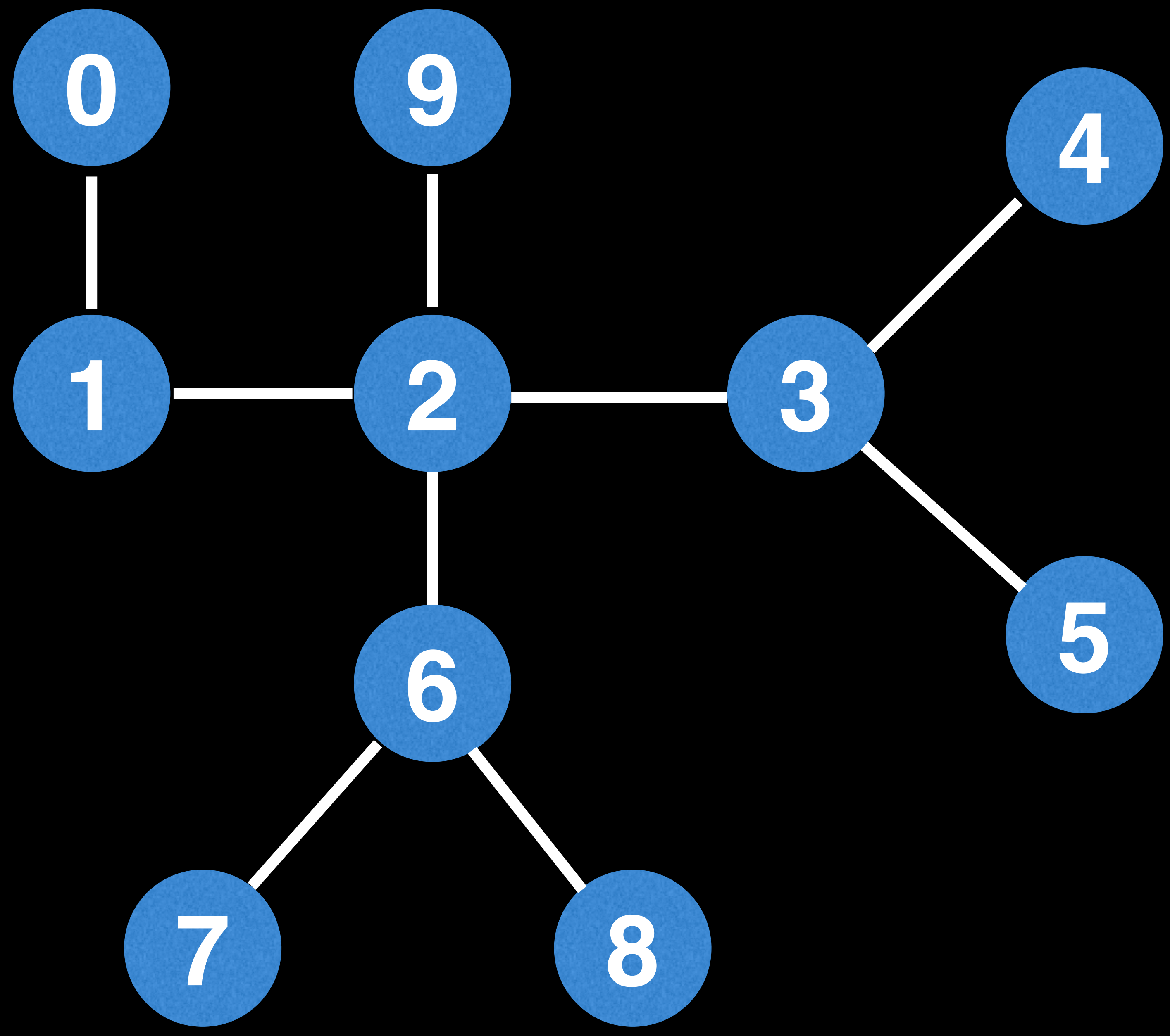
Notice that the center is always the middle vertex or middle two vertices in every longest path along the tree.

Center(s) of undirected tree



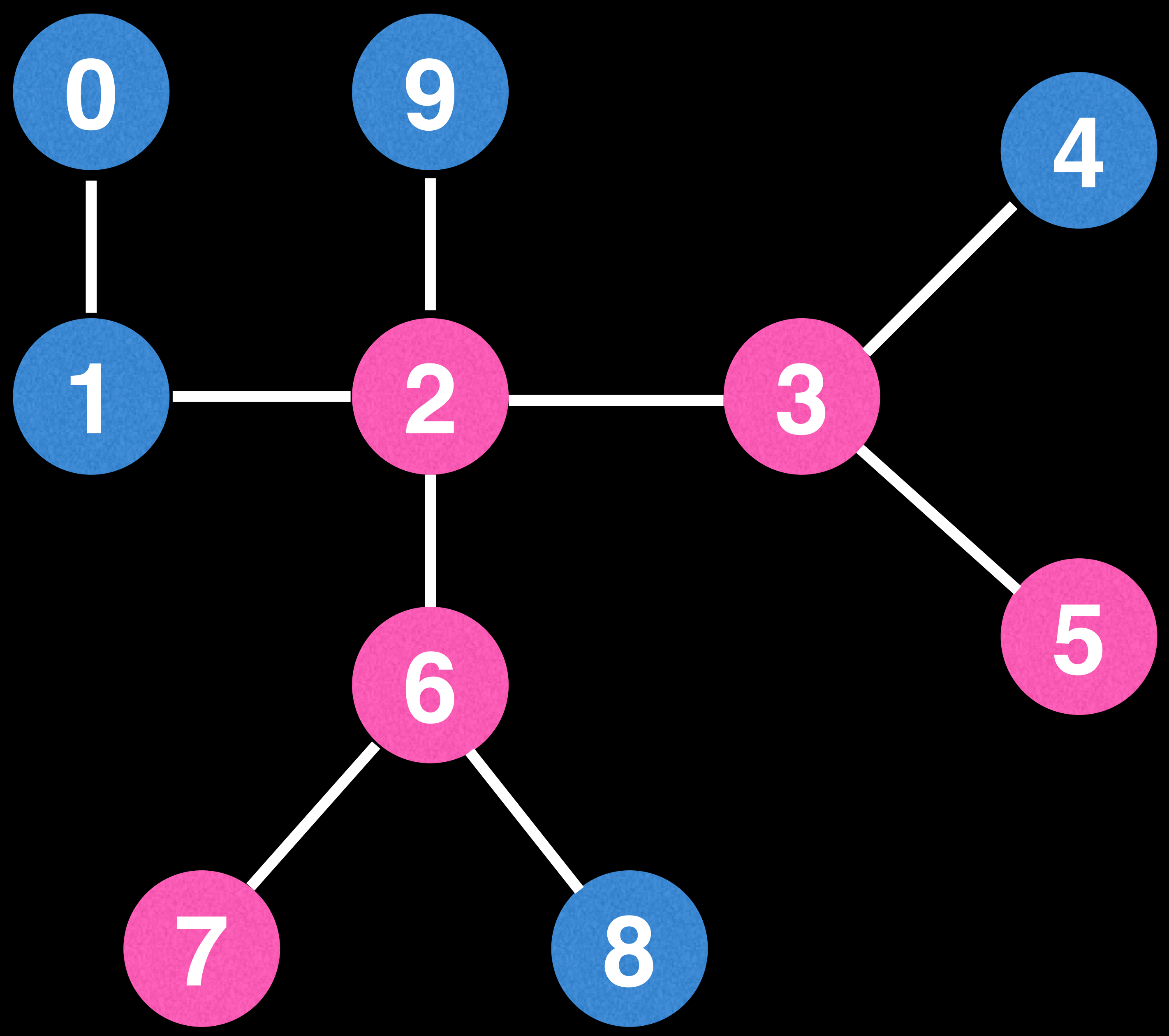
Notice that the center is always the middle vertex or middle two vertices in every longest path along the tree.

Center(s) of undirected tree



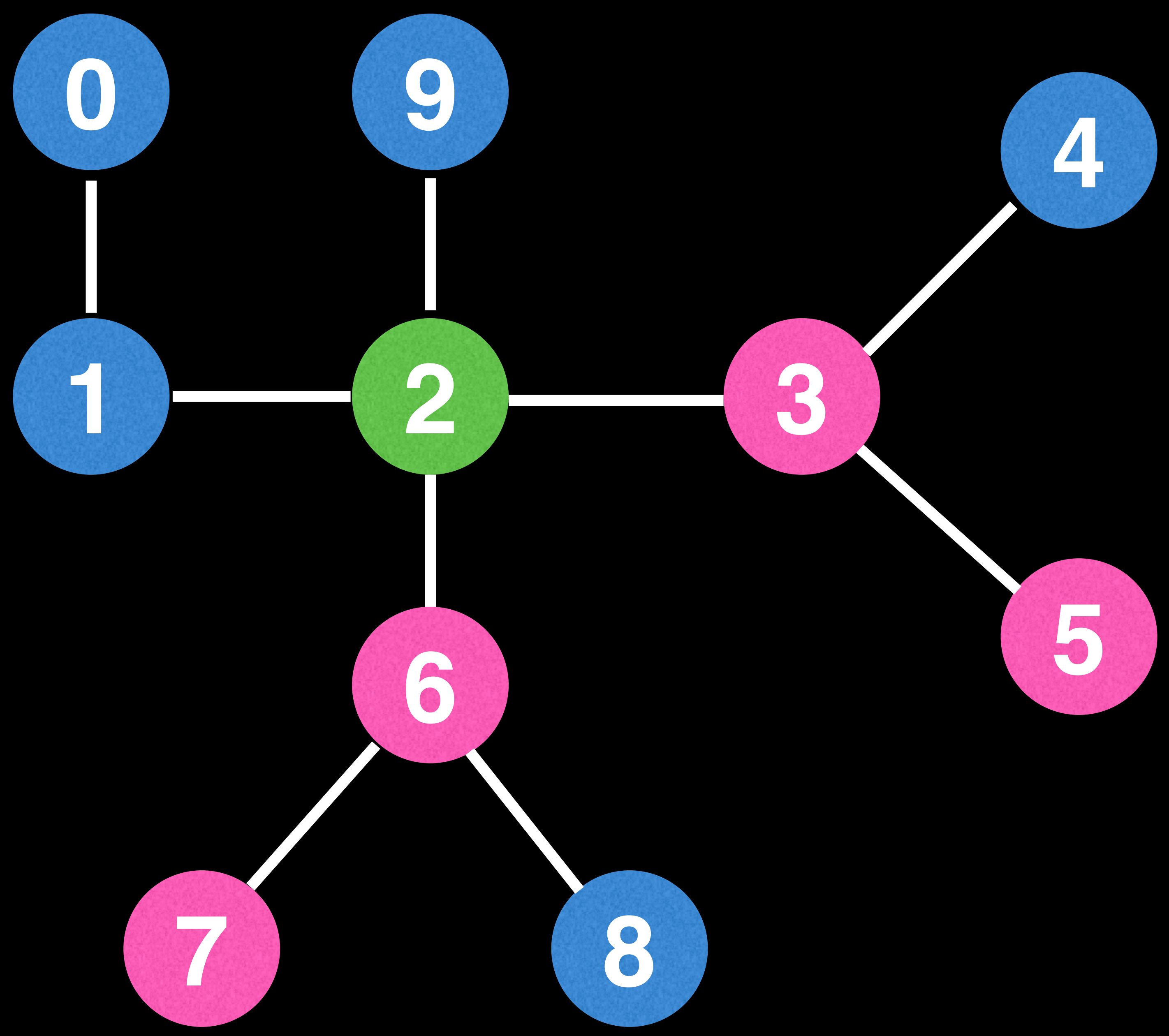
Notice that the center is always the middle vertex or middle two vertices in every longest path along the tree.

Center(s) of undirected tree



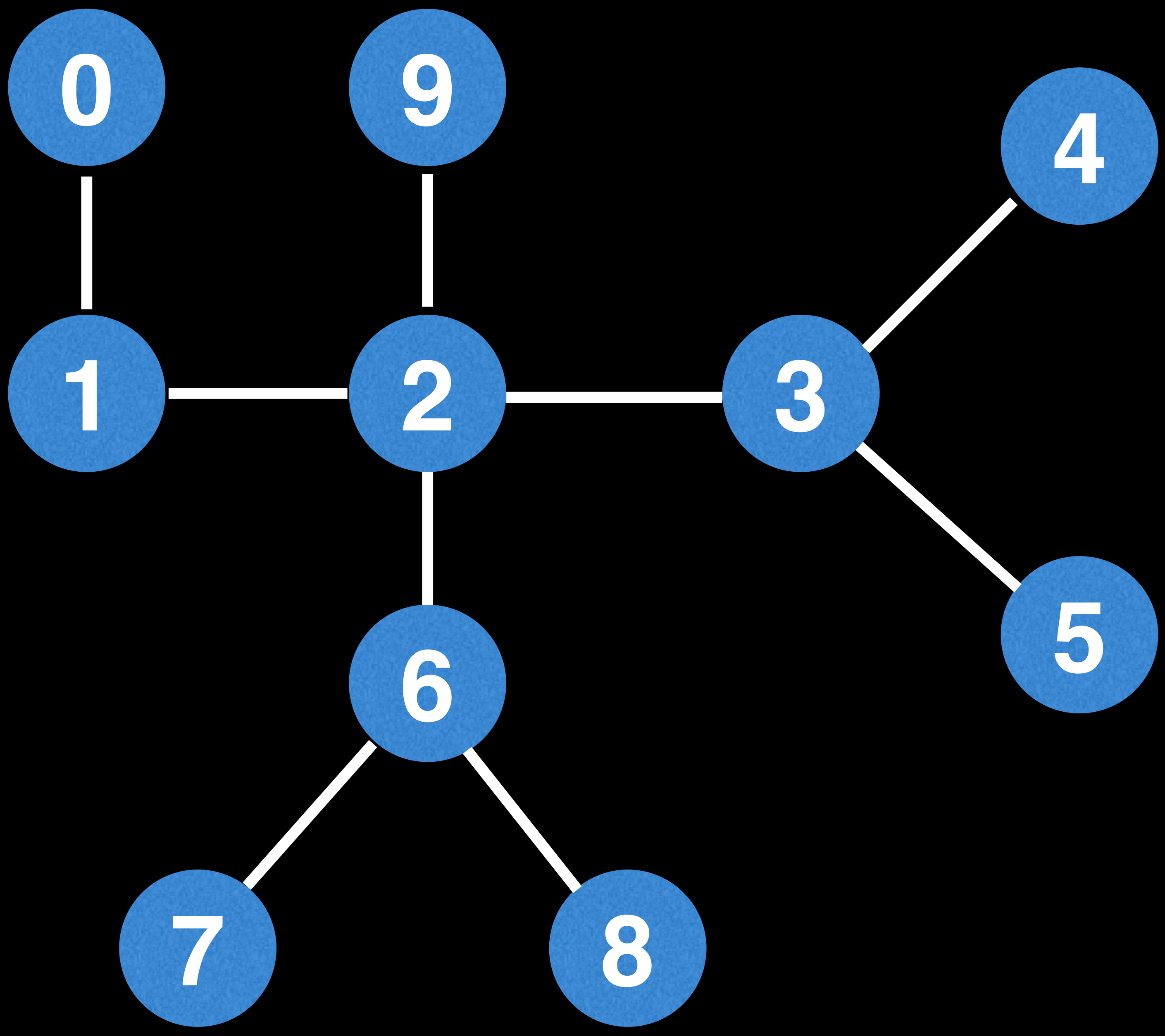
Notice that the center is always the middle vertex or middle two vertices in every longest path along the tree.

Center(s) of undirected tree



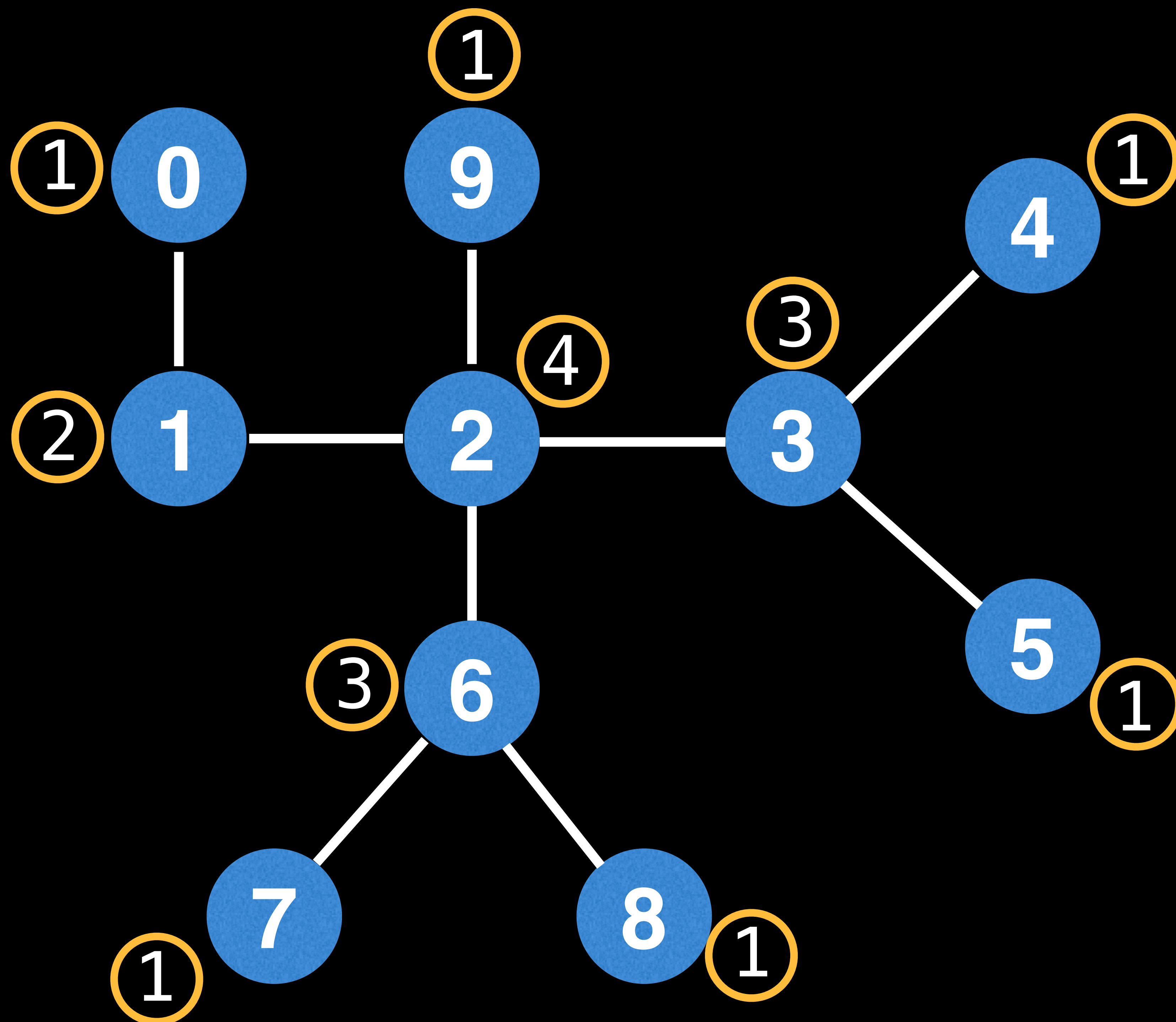
Notice that the center is always the middle vertex or middle two vertices in every longest path along the tree.

Center(s) of undirected tree



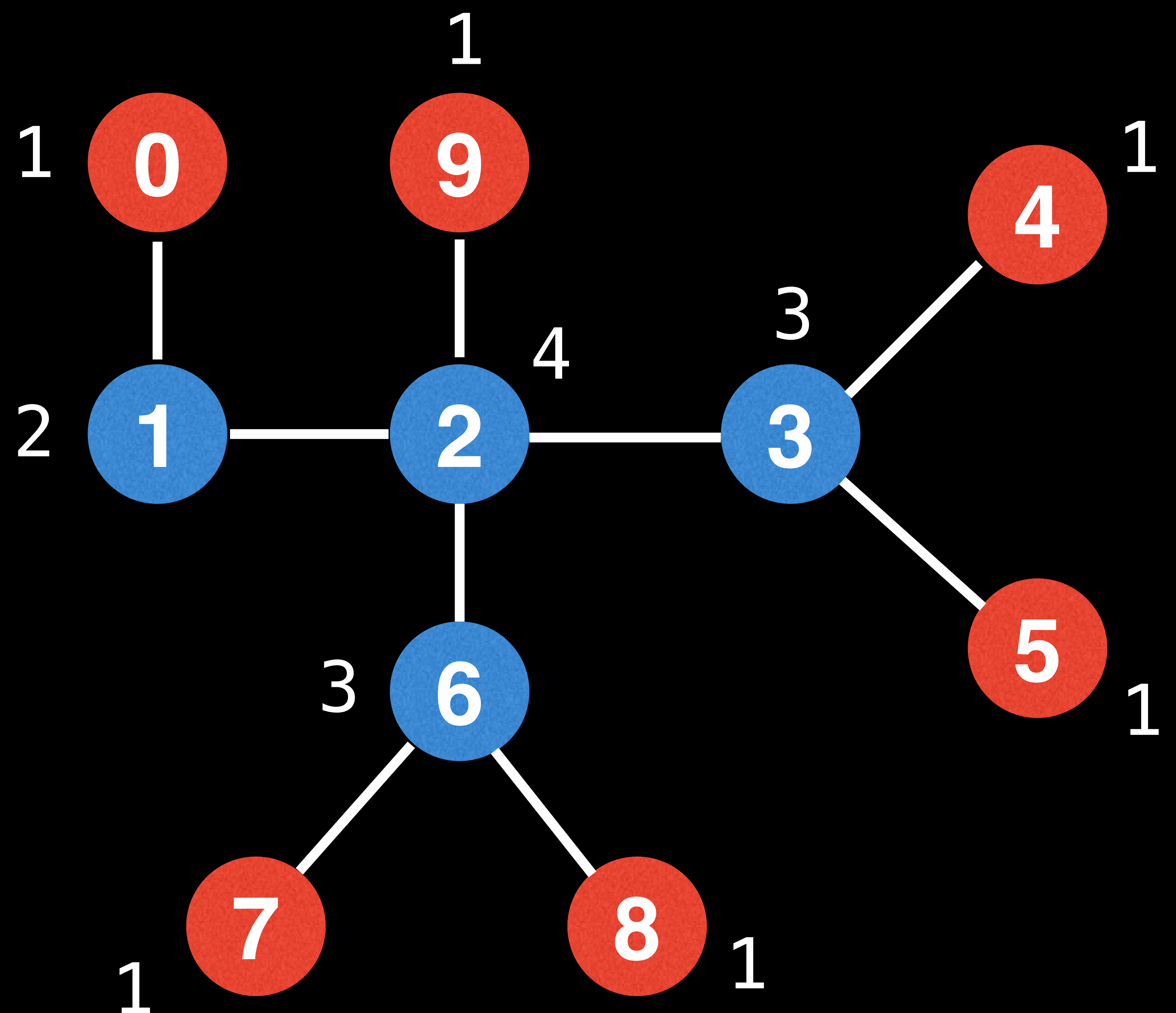
Another approach to find the center is to iteratively pick off each leaf node layer like we were peeling an onion.

Center(s) of undirected tree

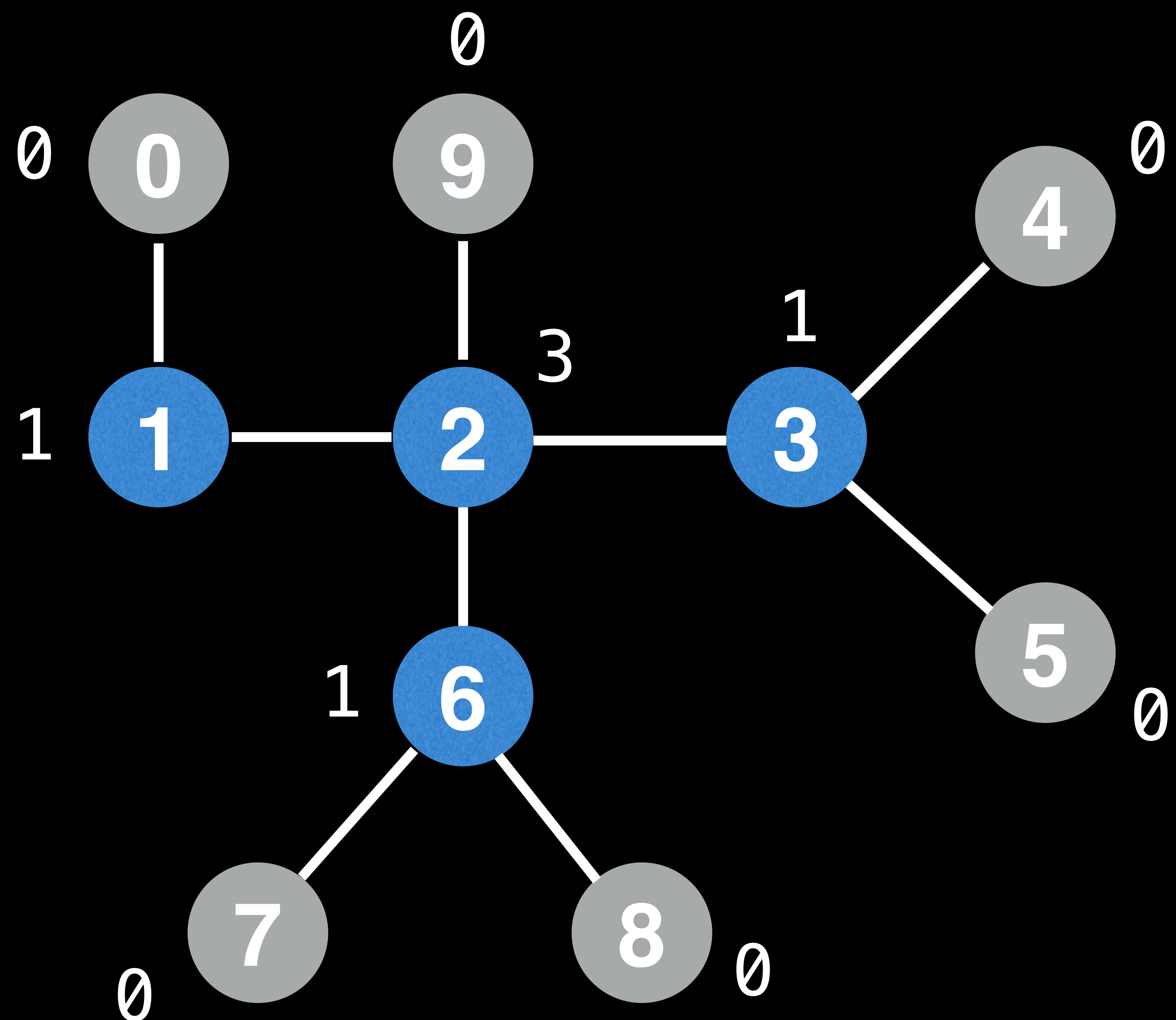


The orange circles represent the **degree** of each node. Observe that each leaf node will have a degree of 1.

Center(s) of undirected tree

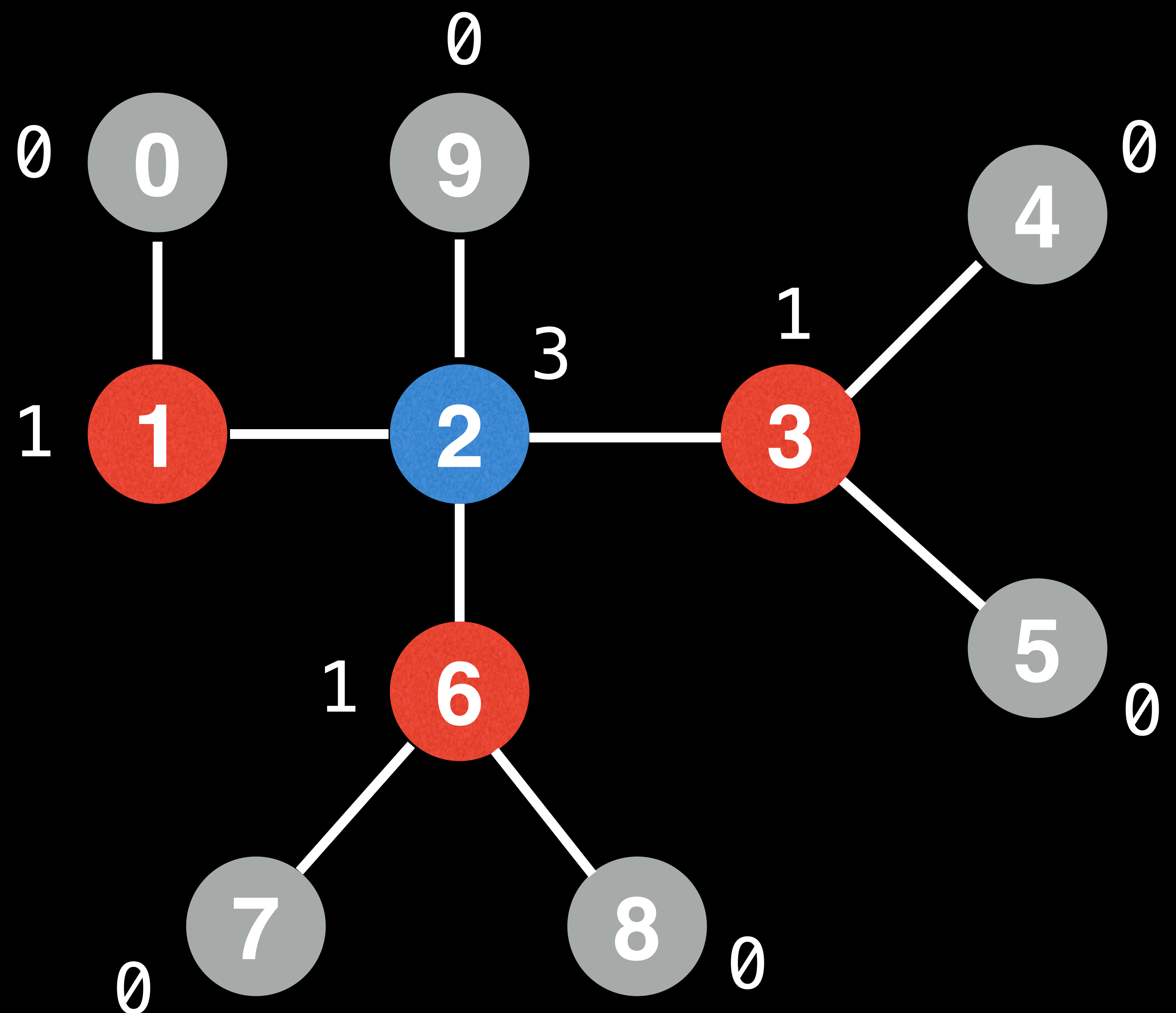


Center(s) of undirected tree

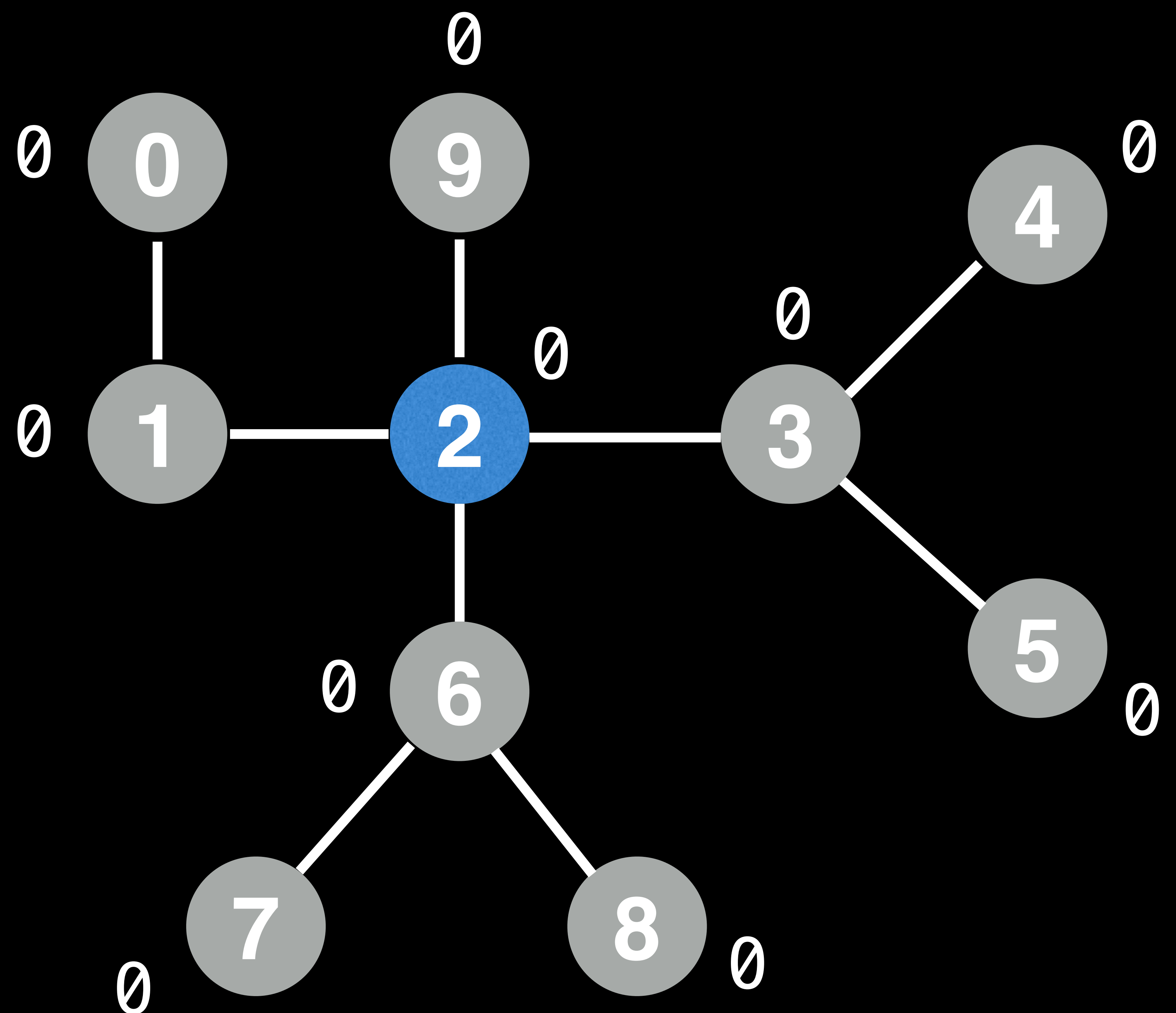


As we prune nodes also reduce the node degree values.

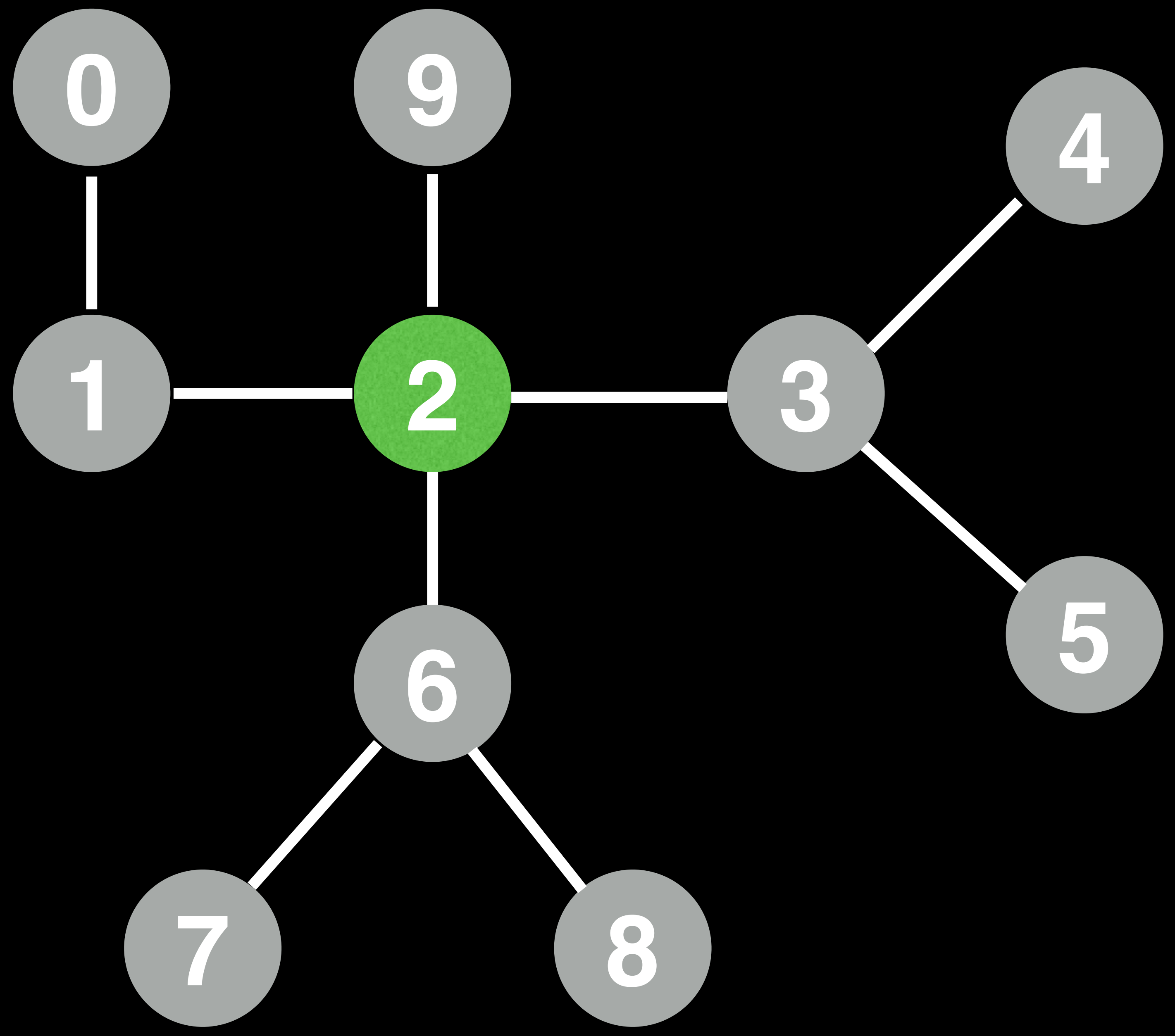
Center(s) of undirected tree



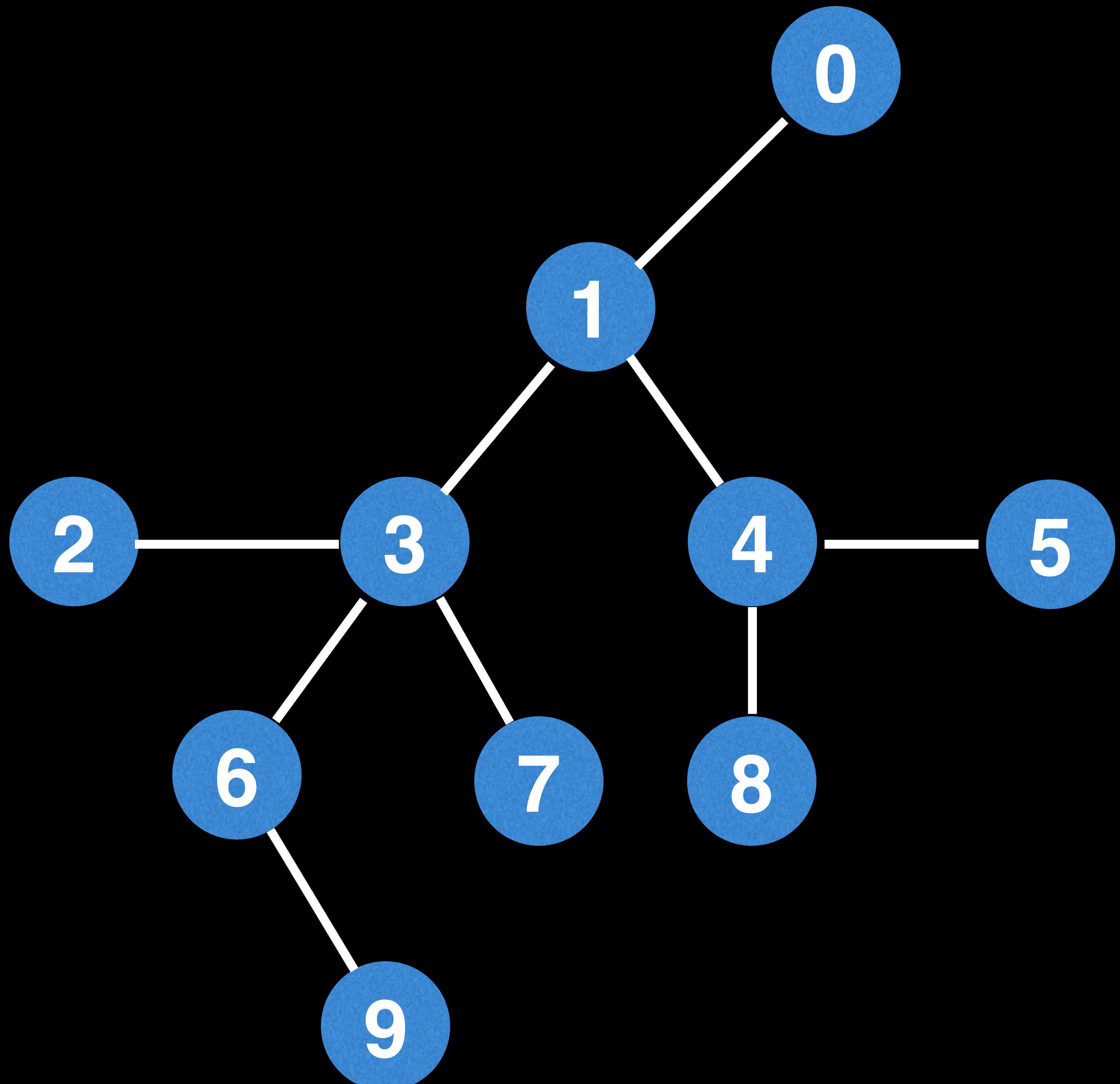
Center(s) of undirected tree



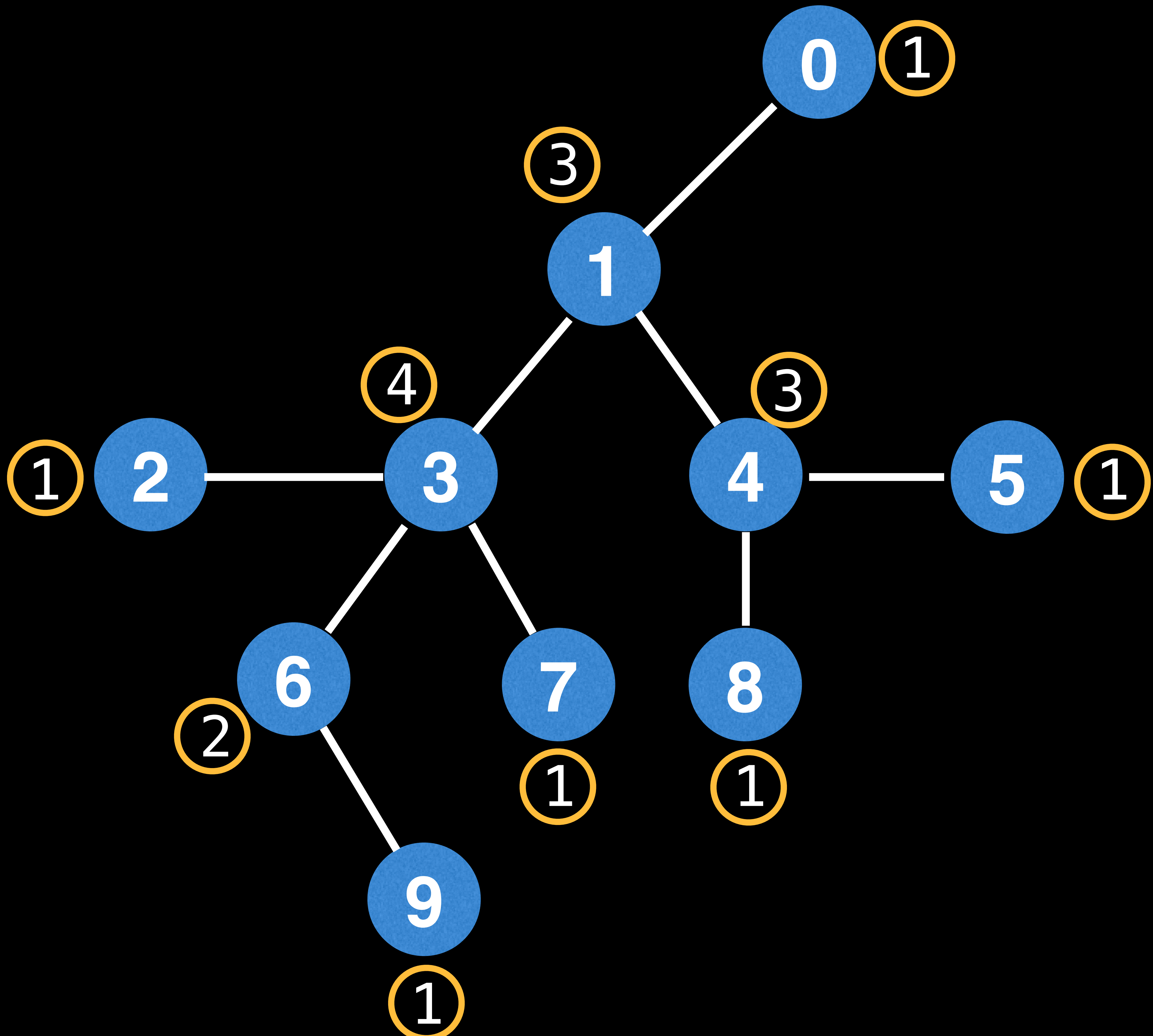
Center(s) of undirected tree



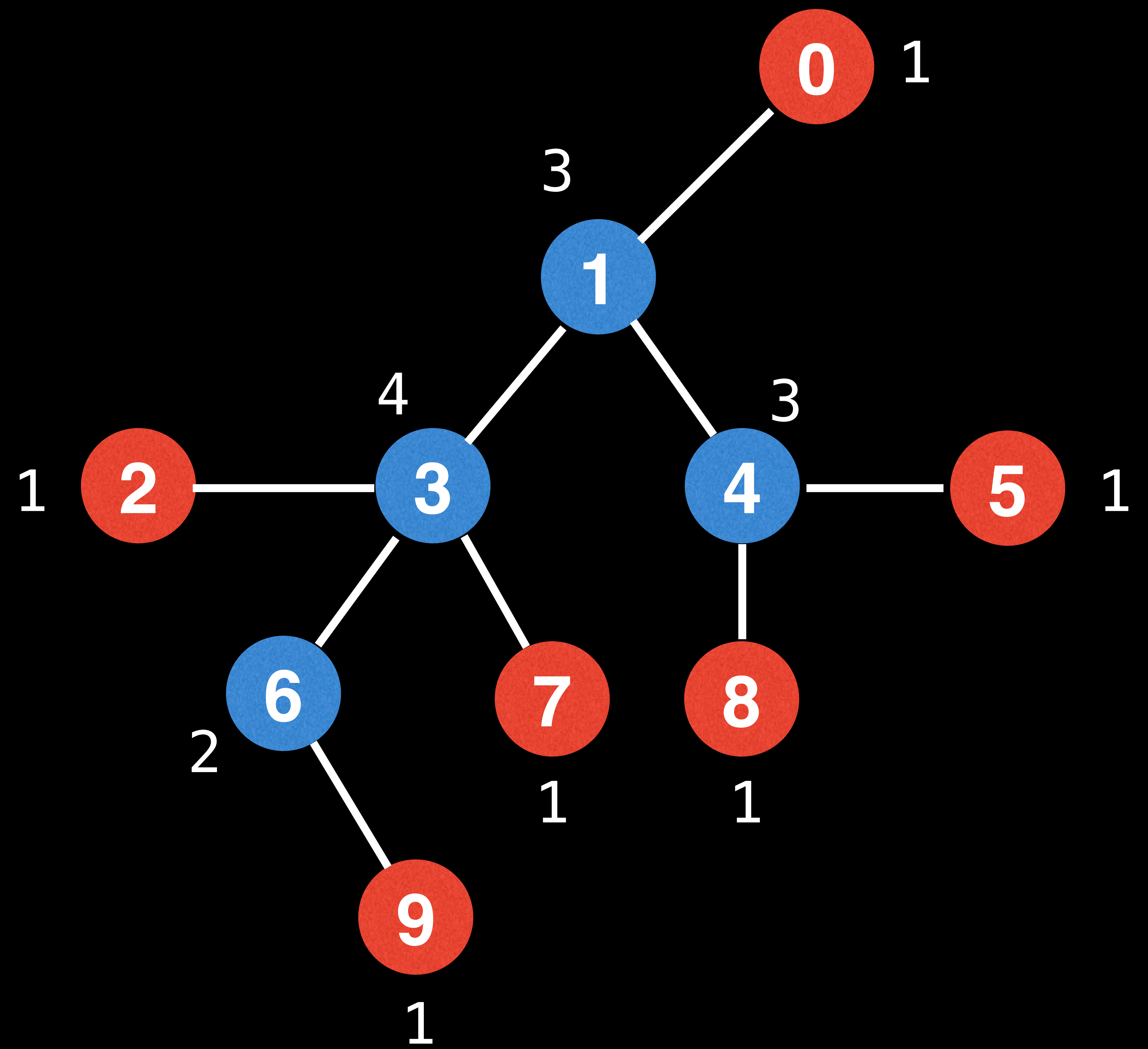
Center(s) of undirected tree



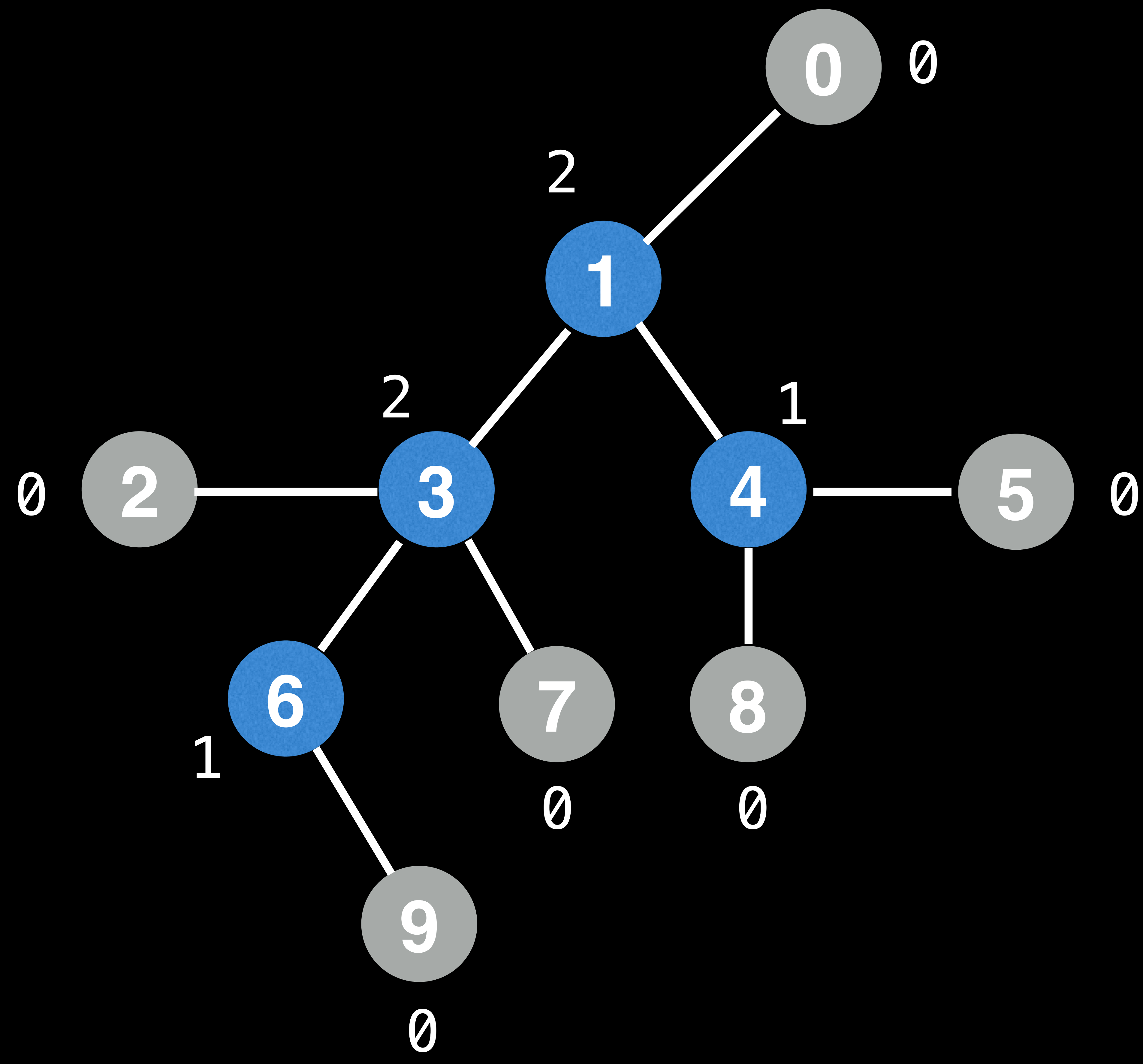
Center(s) of undirected tree



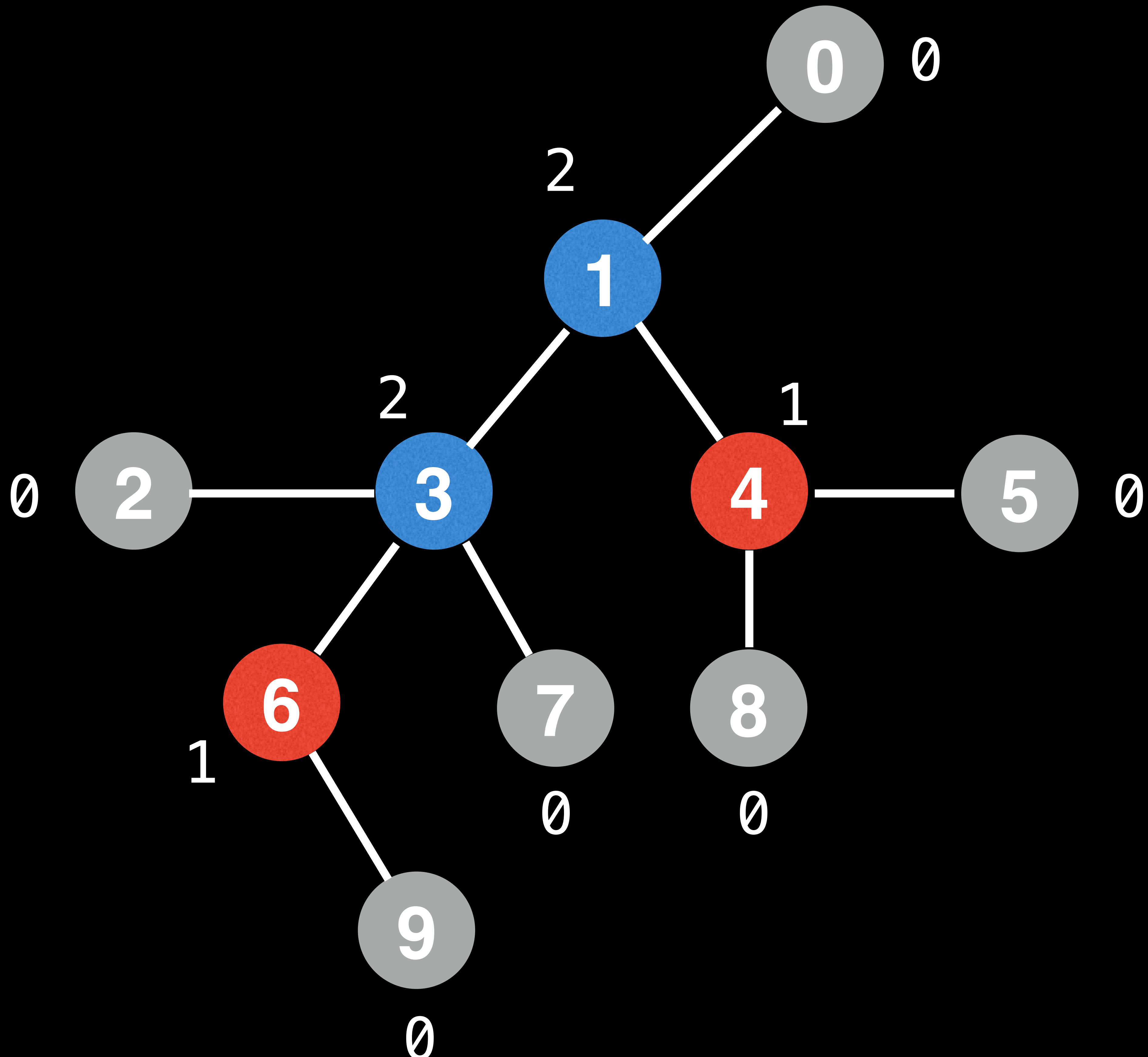
Center(s) of undirected tree



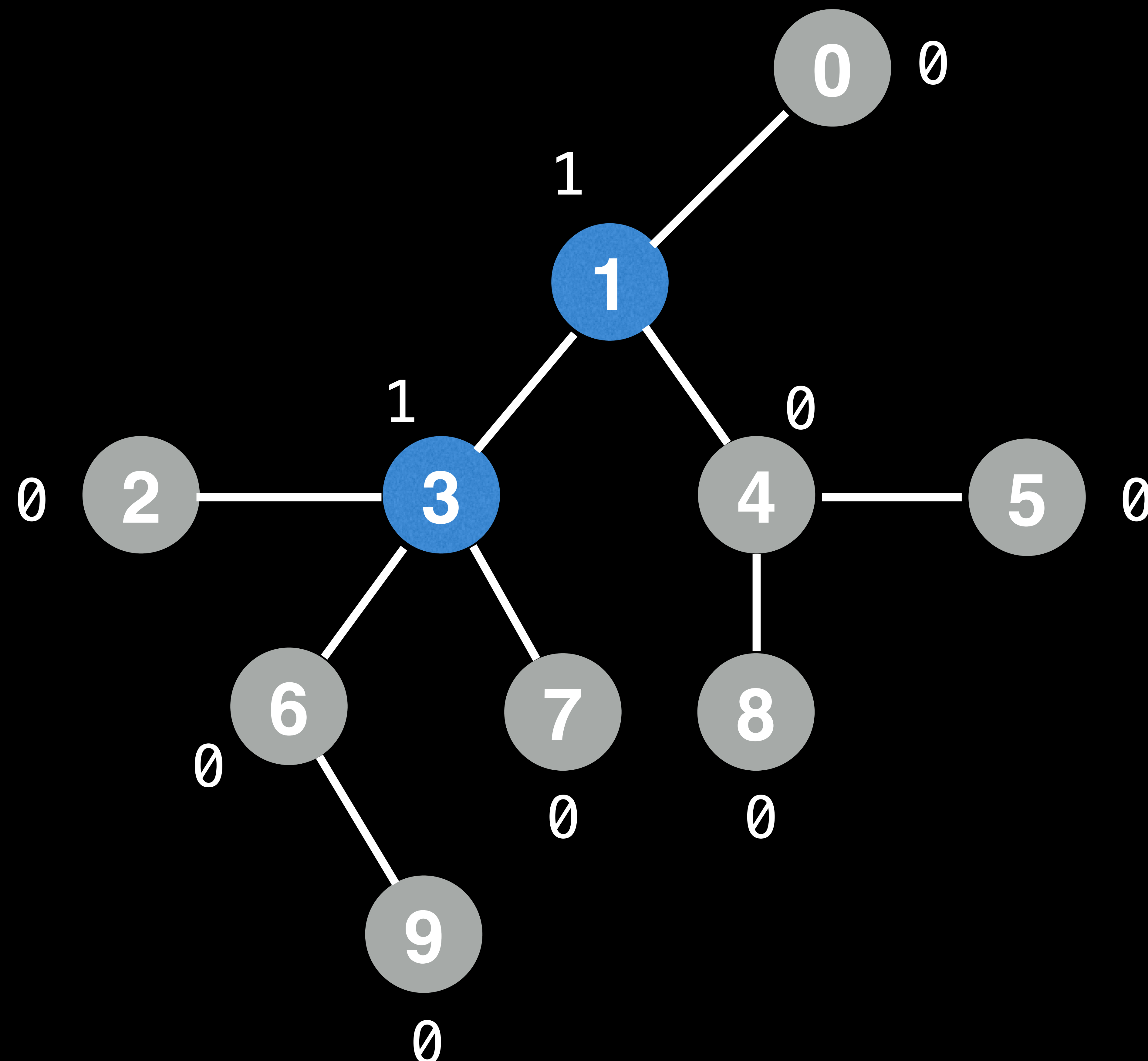
Center(s) of undirected tree



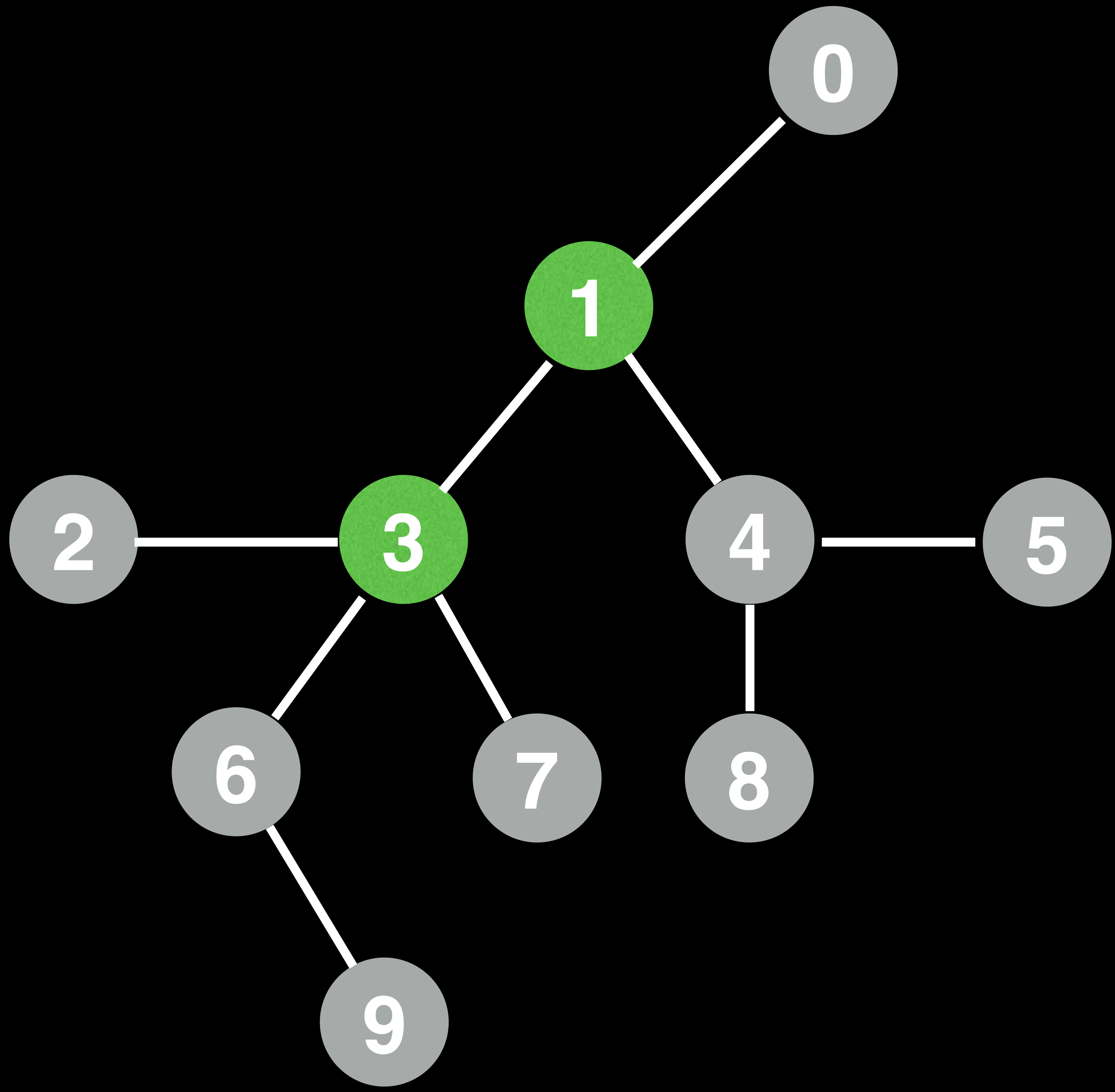
Center(s) of undirected tree



Center(s) of undirected tree



Center(s) of undirected tree



Some trees have two centers

```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] – 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] – 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] - 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] - 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] - 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] - 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] - 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] - 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] - 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

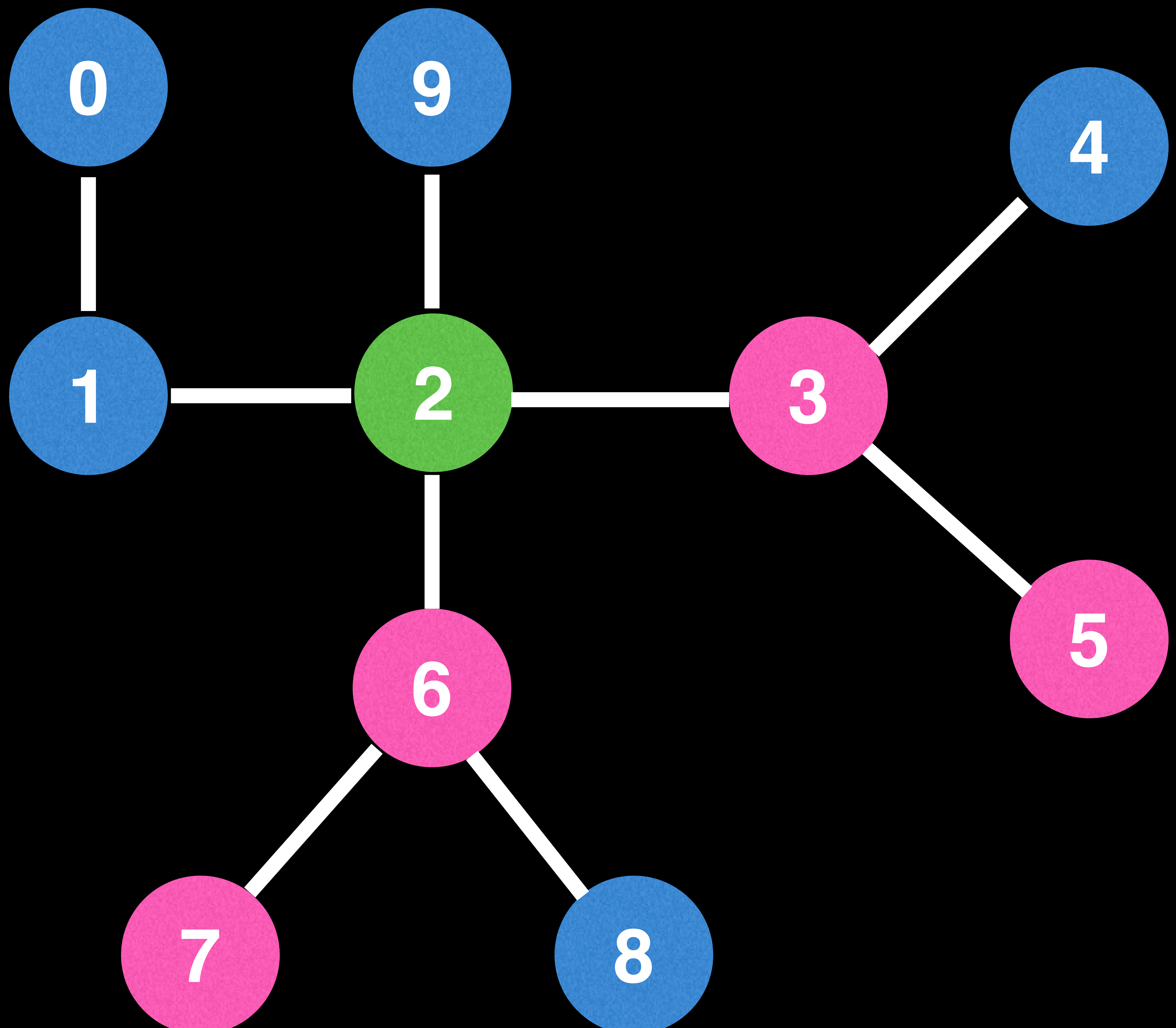
```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] - 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

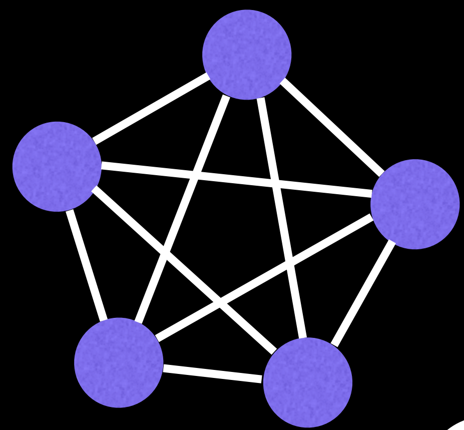
```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] - 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

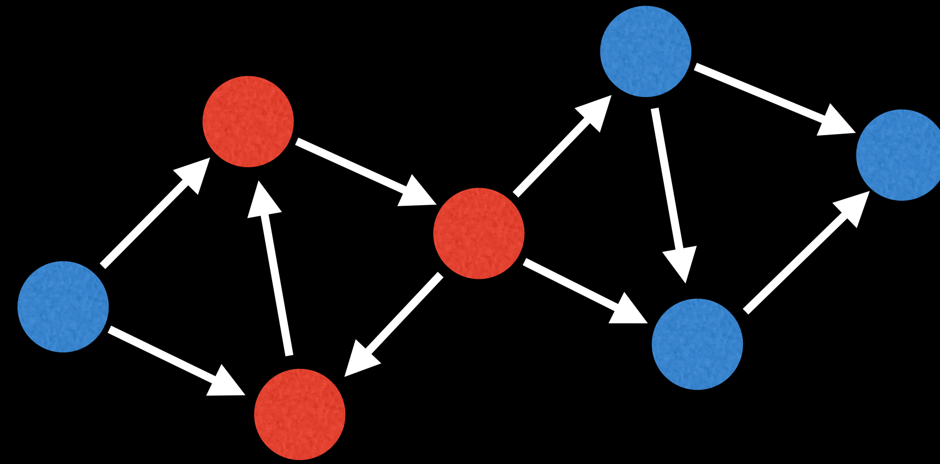
```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] - 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

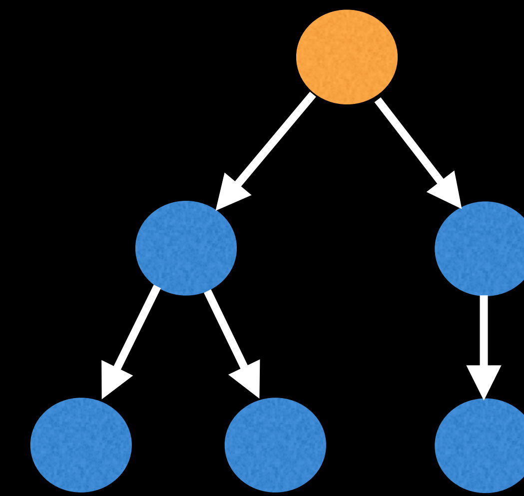
```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] - 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

Center(s) of a Tree

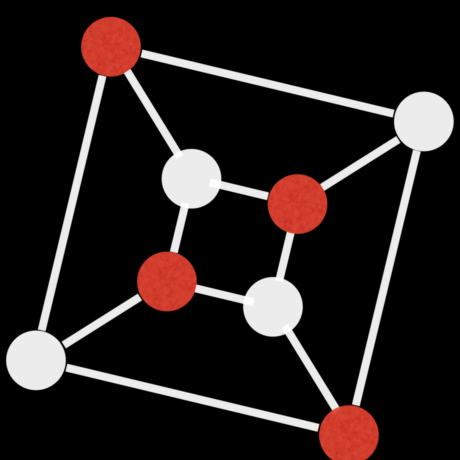


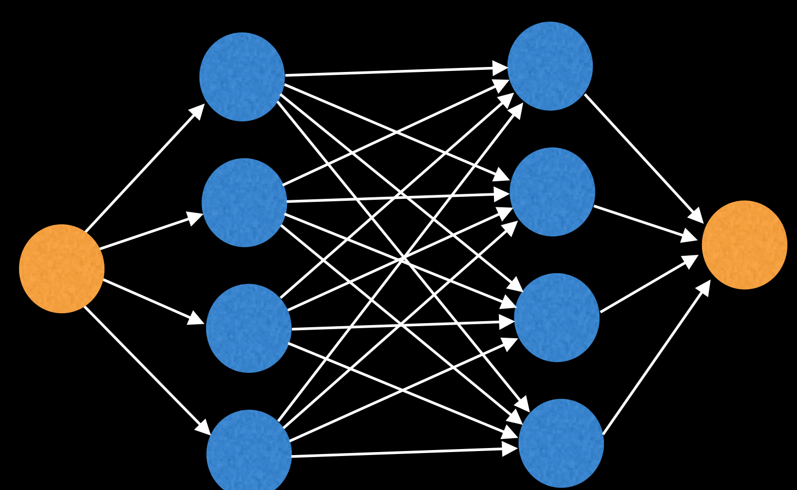


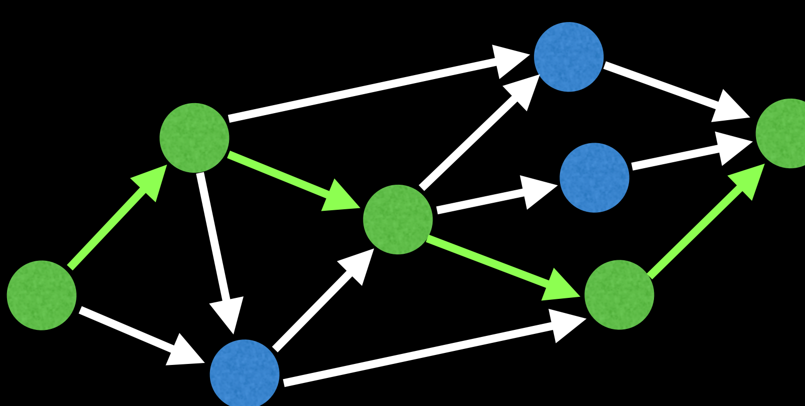




Graph Theory Video Series







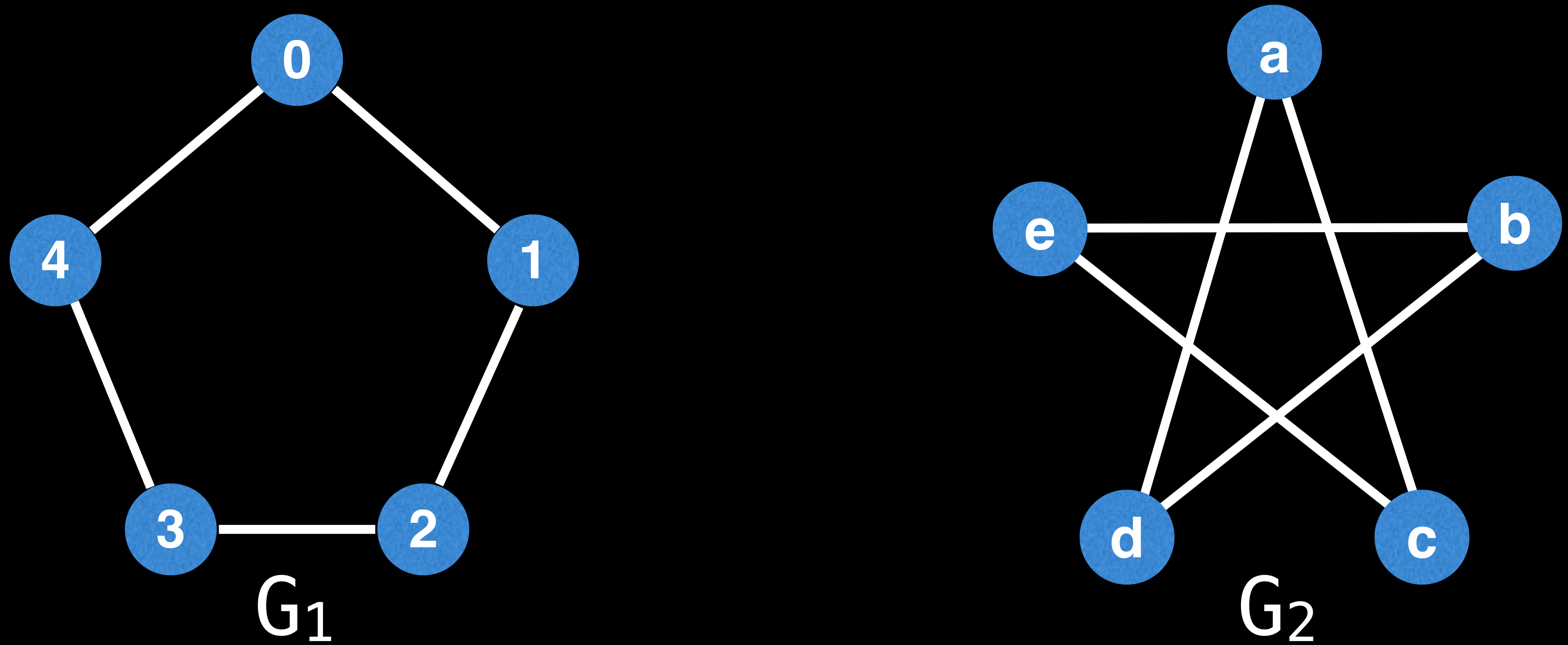
Isomorphisms in trees

A question of equality

William Fiset

Graph Isomorphism

The question of asking whether two graphs G_1 and G_2 are **isomorphic** is asking whether they are *structurally* the same.



Even though G_1 and G_2 are labelled differently and may appear different they are structurally the same graph.

Graph Isomorphism

We can also define the notion of a graph isomorphism more rigorously:

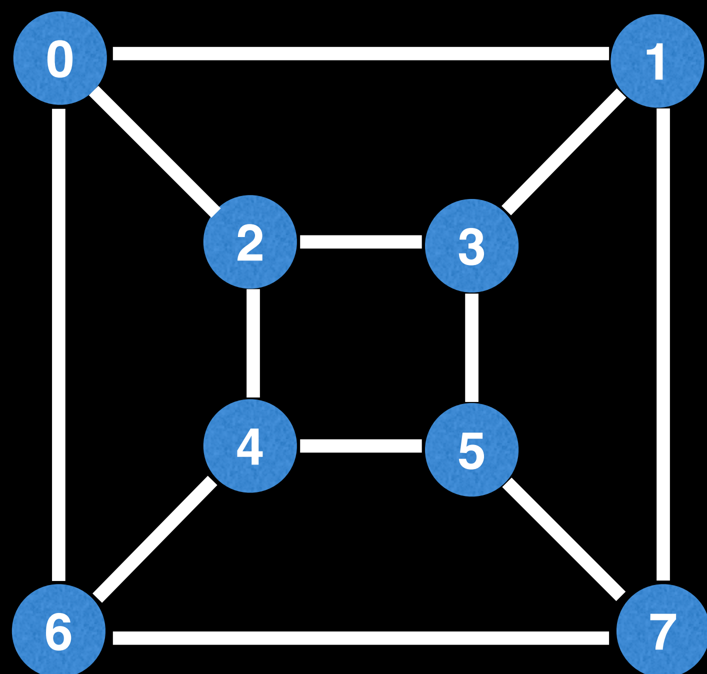
$G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are isomorphic if there exists a **bijection** φ between the sets $V_1 \rightarrow V_2$ such that:

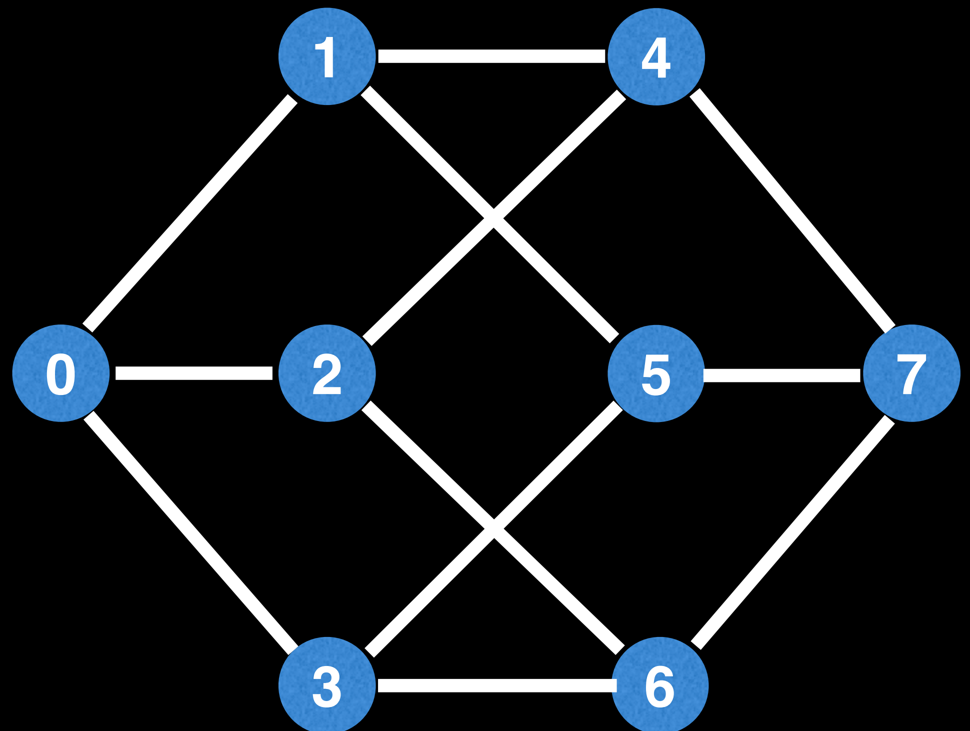
$$\forall\ u,v \in V_1,\ (u,v) \in E_1 \iff (\varphi(u),\varphi(v)) \in E_2$$

In simple terms, for an isomorphism to exist there needs to be a function φ which can map all the nodes/edges in G_1 to G_2 and vice-versa.

Graph Isomorphism

Determining if two graphs are isomorphic is not only not obvious to the human eye, but also a difficult problem for computers.

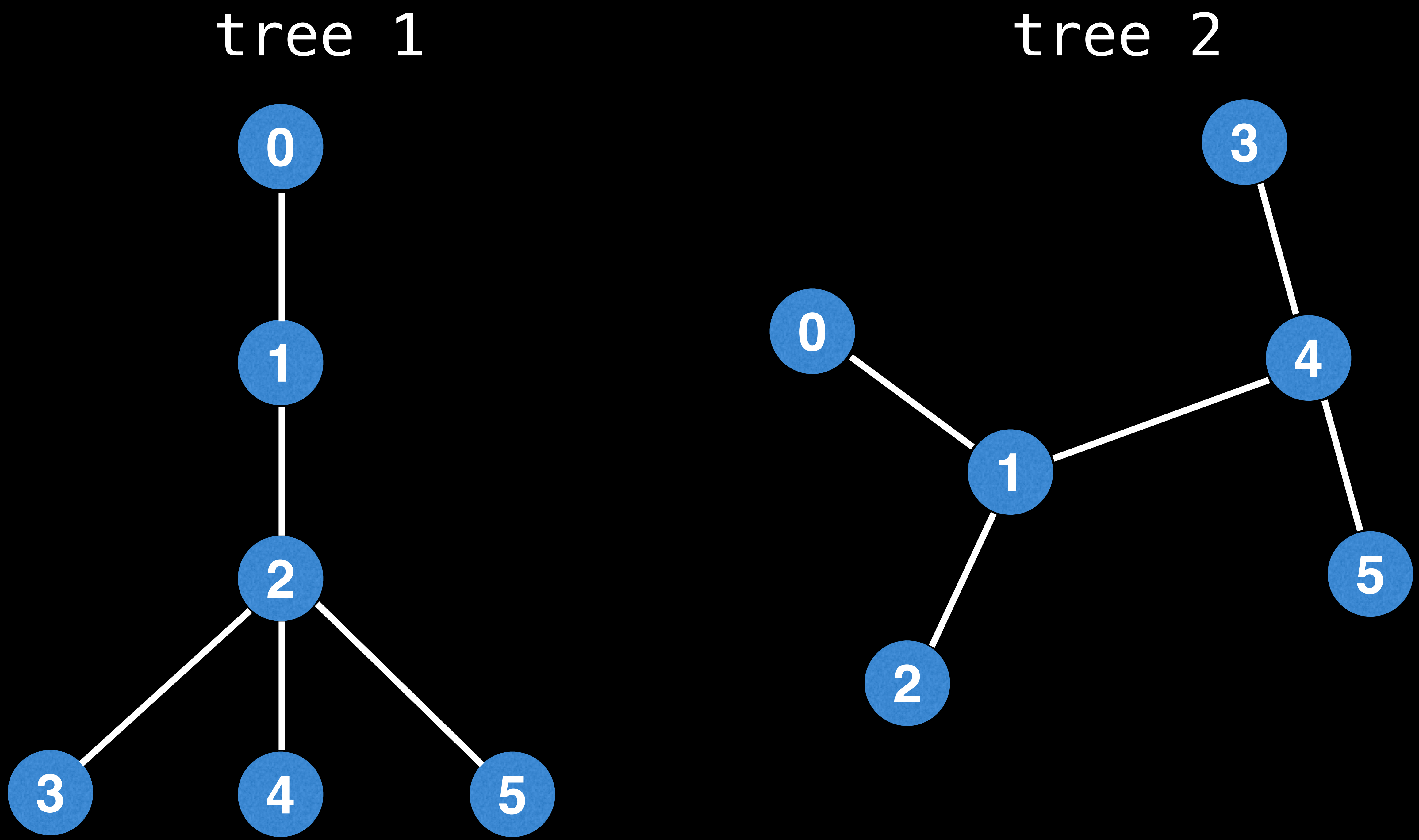




It is still an open question as to whether the graph isomorphism problem is NP complete. However, many polynomial time isomorphism algorithms exist for graph subclasses such as trees.

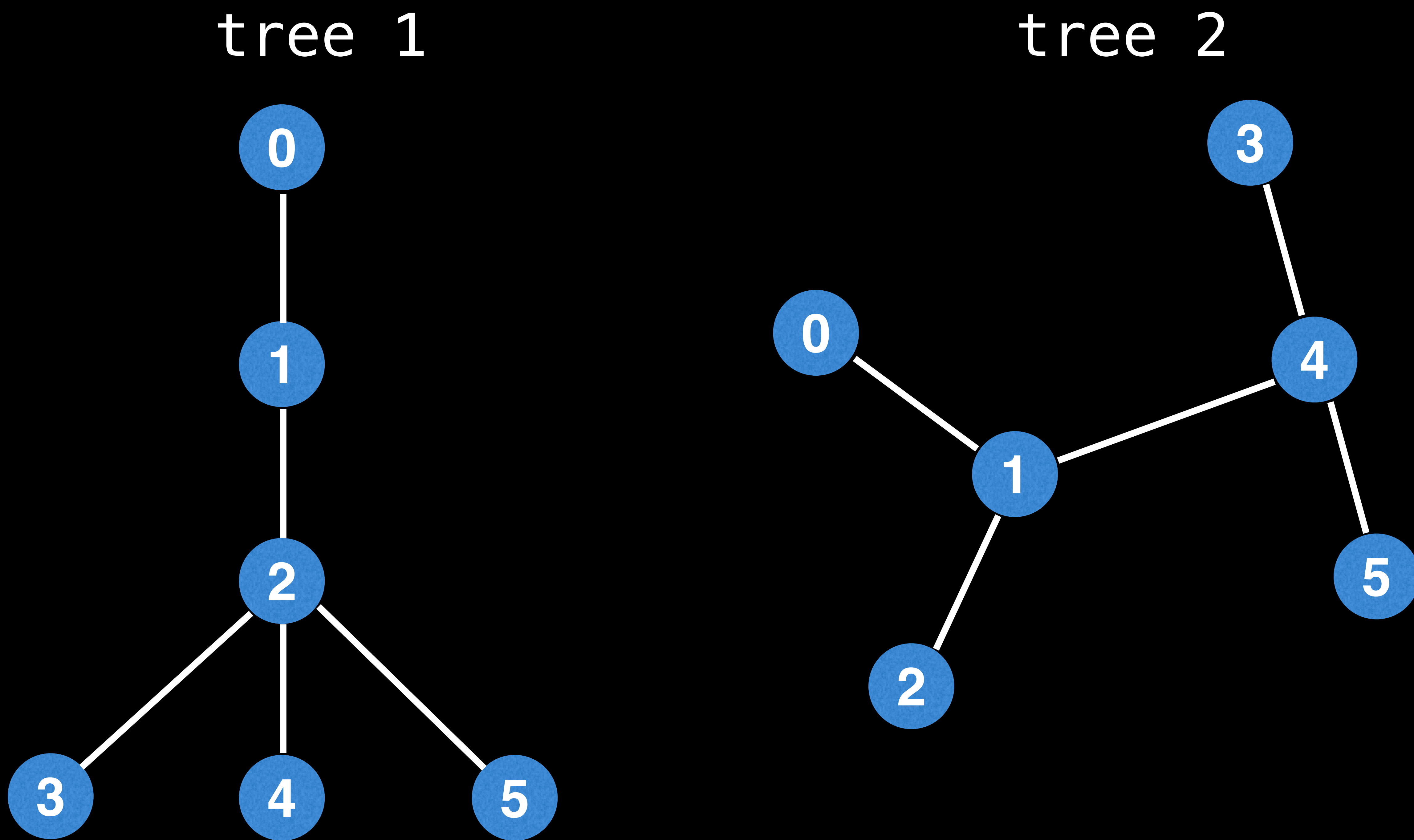
Isomorphic Trees

Isomorphic Trees



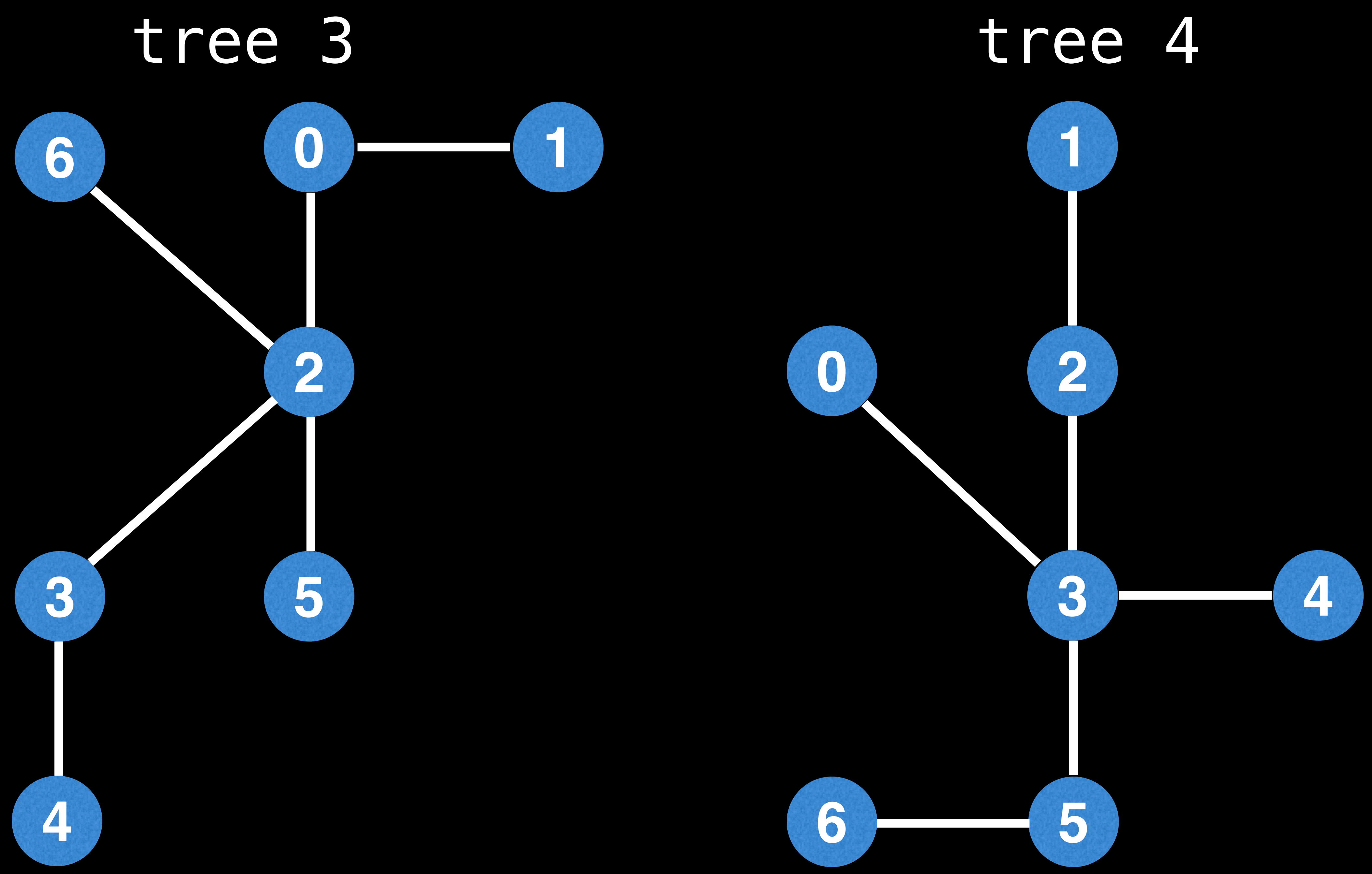
Q: Are these trees isomorphic?

Isomorphic Trees



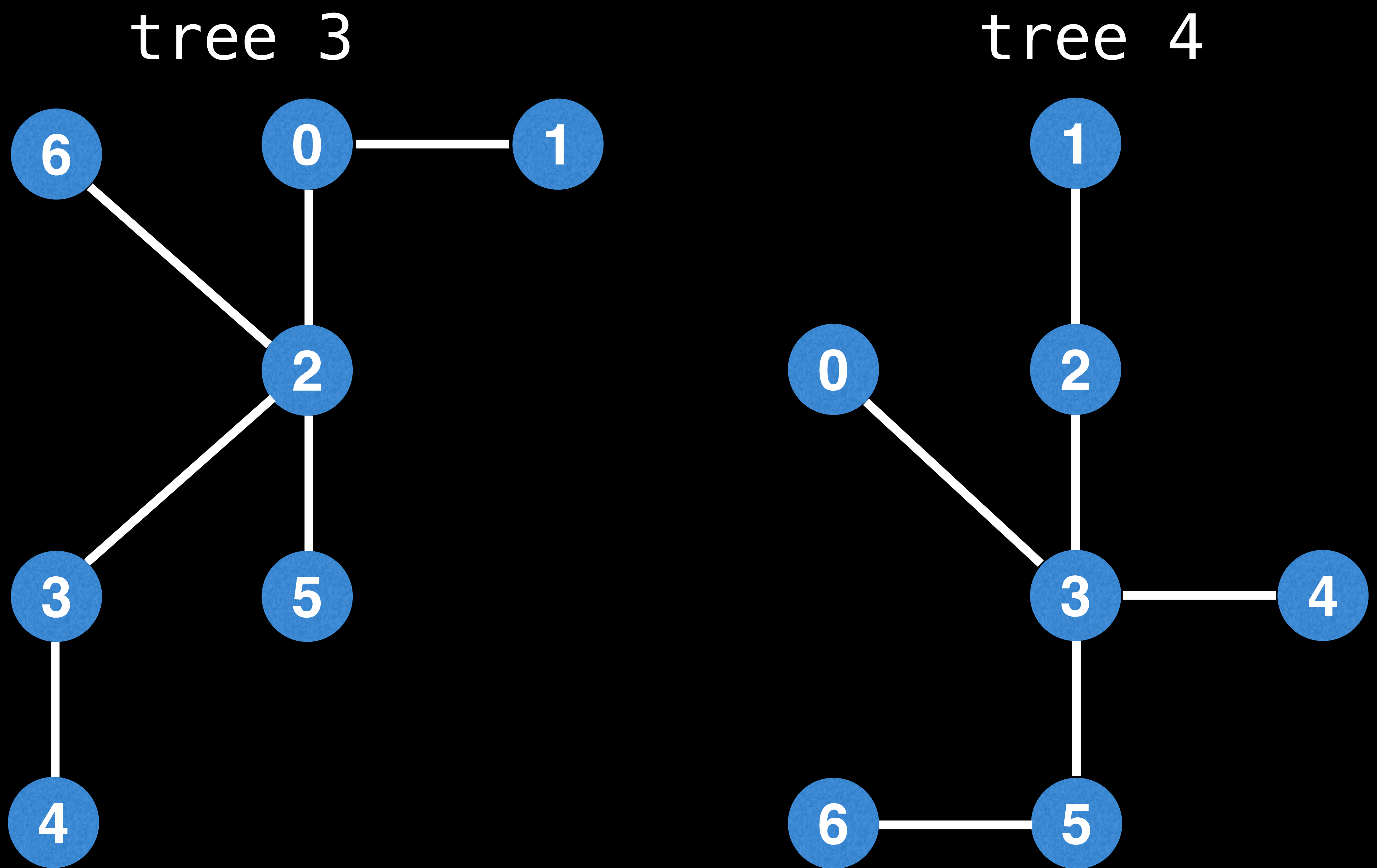
A: no, these trees are structurally different.

Isomorphic Trees



Q: Are these trees isomorphic?

Isomorphic Trees



Yes, one possible label mapping is:
6->0, 1->1, 0->2, 2->3, 5->4, 3->5, 4->6

Identifying Isomorphic Trees

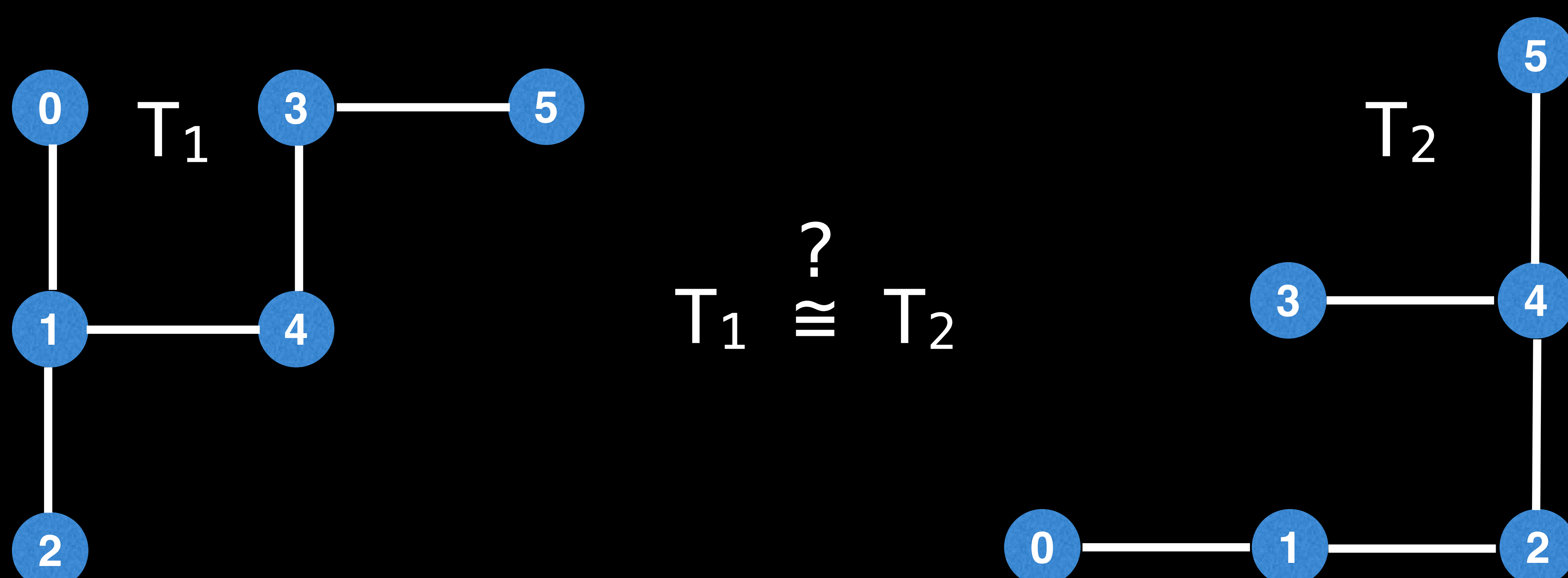
There are several very quick **probabilistic** (usually hash or heuristic based) algorithms for identifying isomorphic trees. These tend to be fast, but also error prone due to hash collisions in a limited integer space.

The method we'll be looking at today involves **serializing** a tree into a **unique encoding**. This unique encoding is simply a unique string that represents a tree, if another tree has the same encoding then they are isomorphic.

Identifying Isomorphic Trees

We can directly serialize an unrooted tree, but in practice serializing a rooted tree is typically easier code wise.

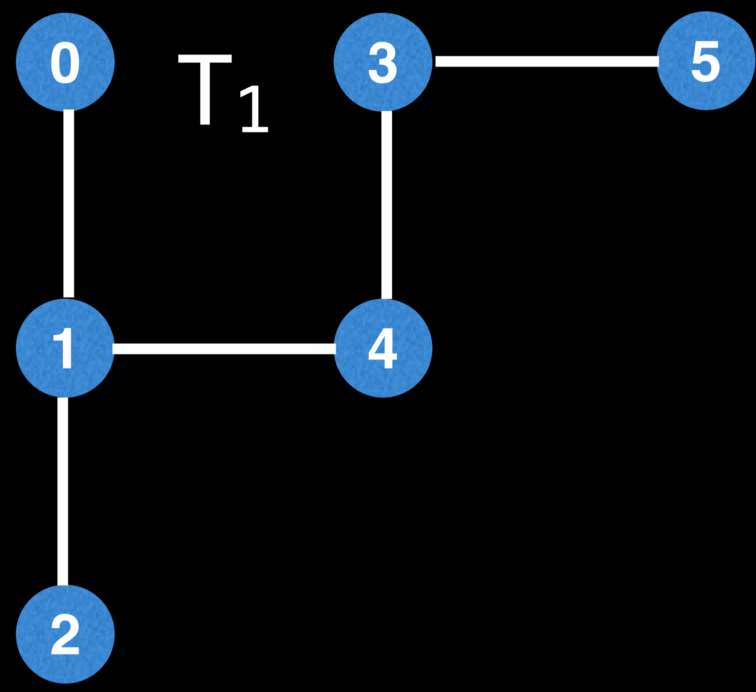
However, one caveat to watch out for if we're going to root our two trees T_1 and T_2 to check if they're isomorphic is to ensure that the same root node is selected in both trees before serializing/encoding the trees.

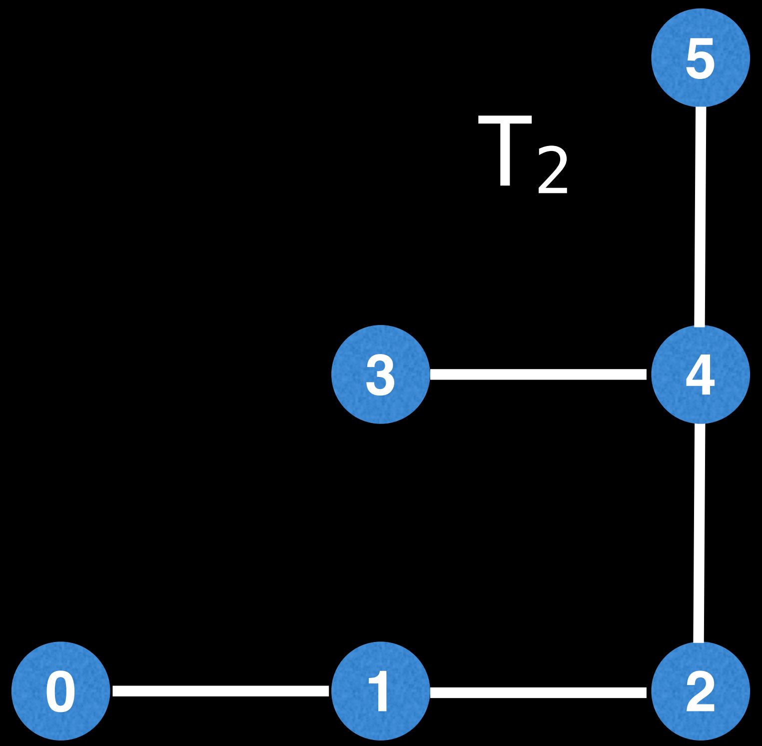


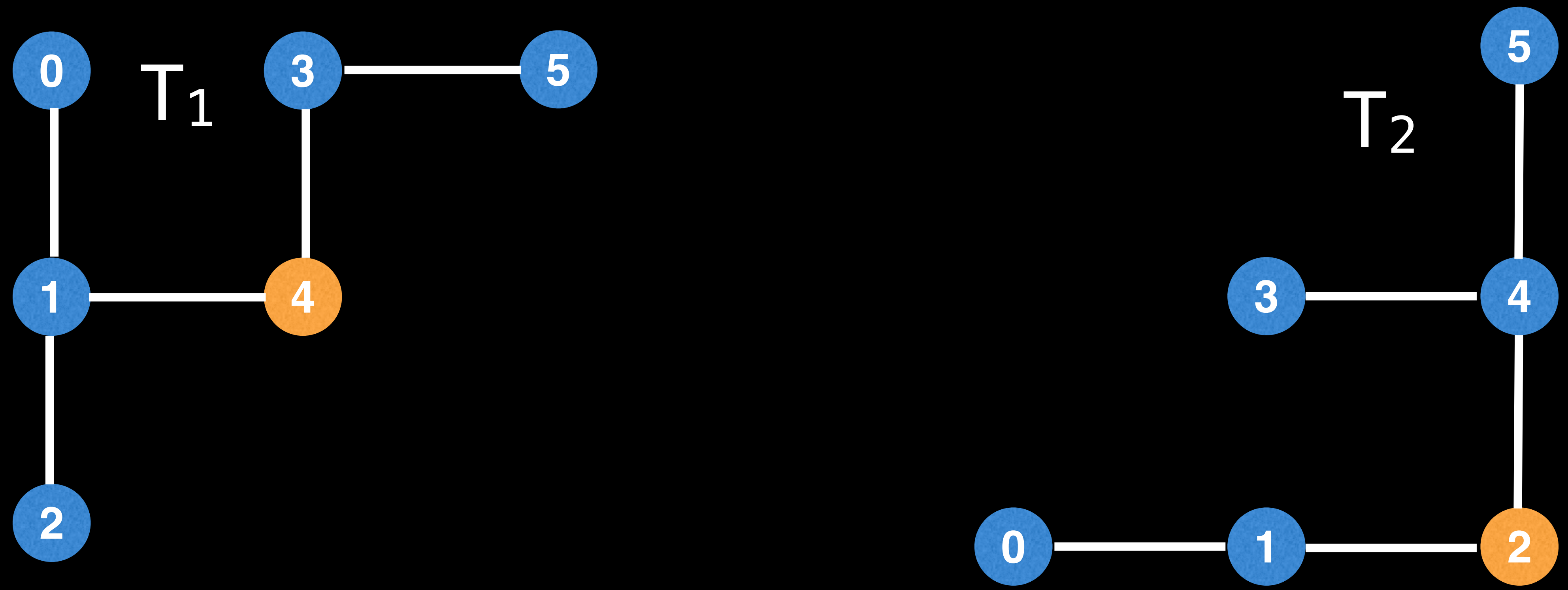
Identifying Isomorphic Trees

To select a common node between both trees we can use what we learned from finding the center(s) of a tree to help ourselves.

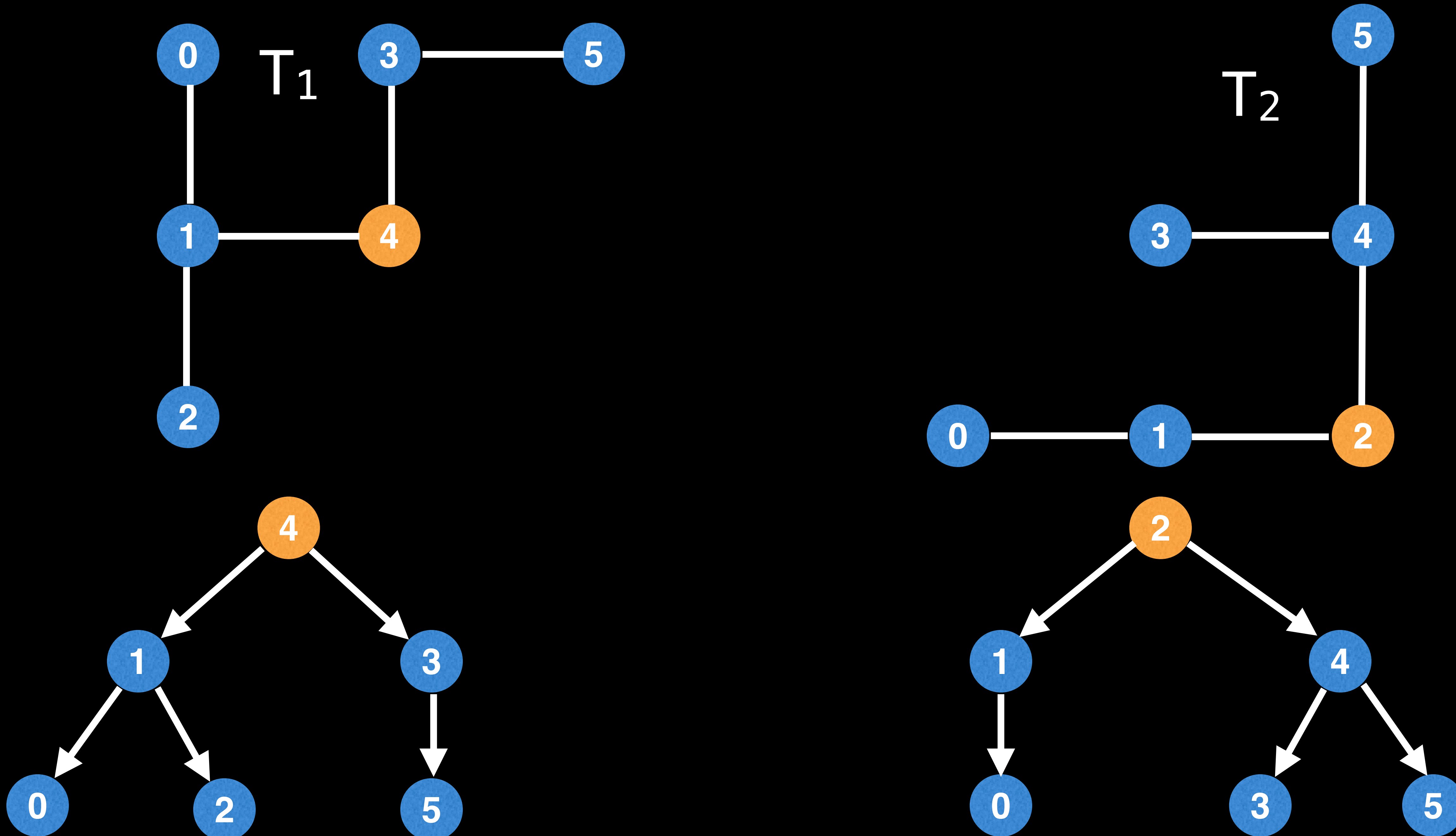
<insert video frame>


T_1
0
3
5
1
4
2

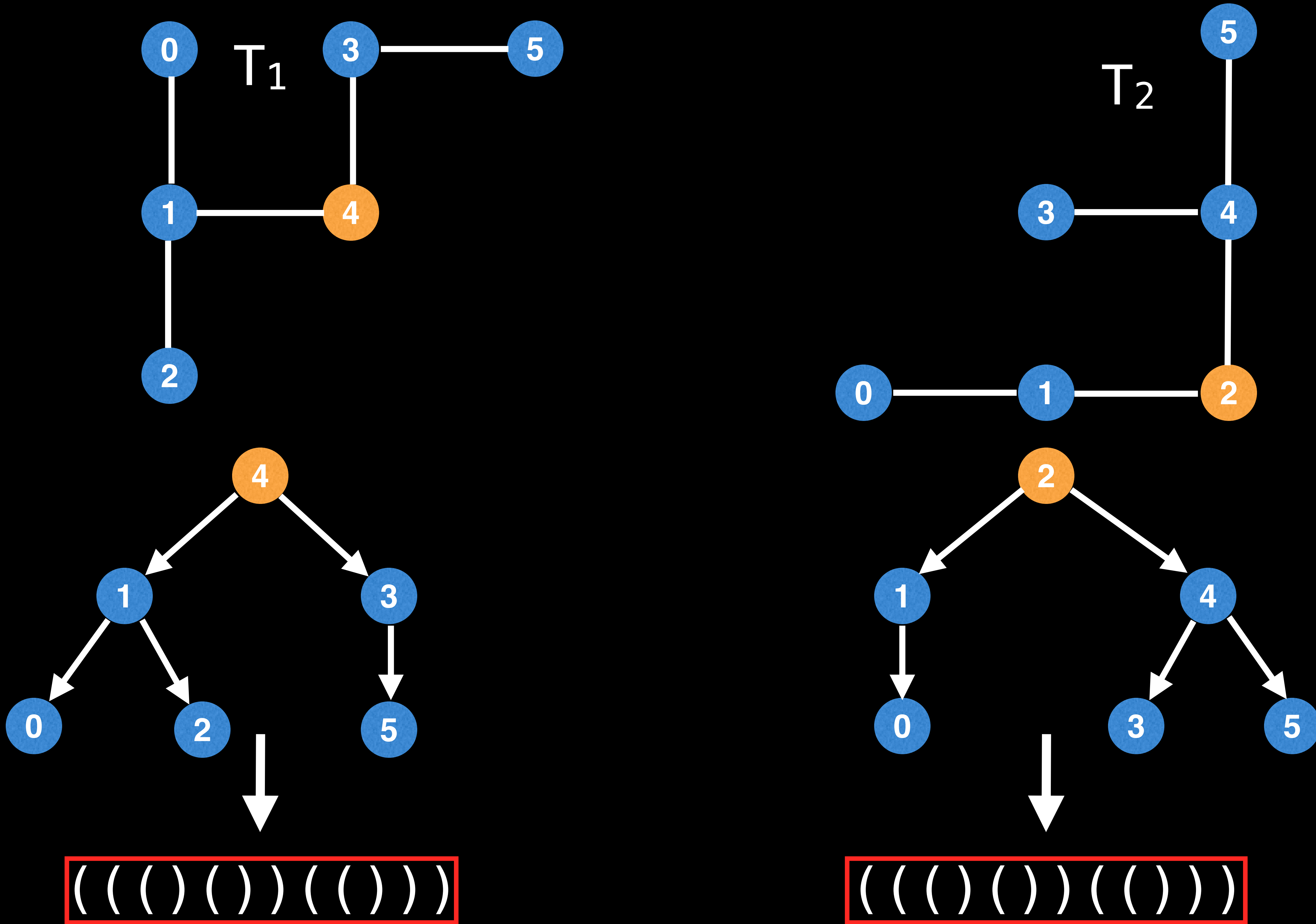

5
T_2
3
4
0
1
2



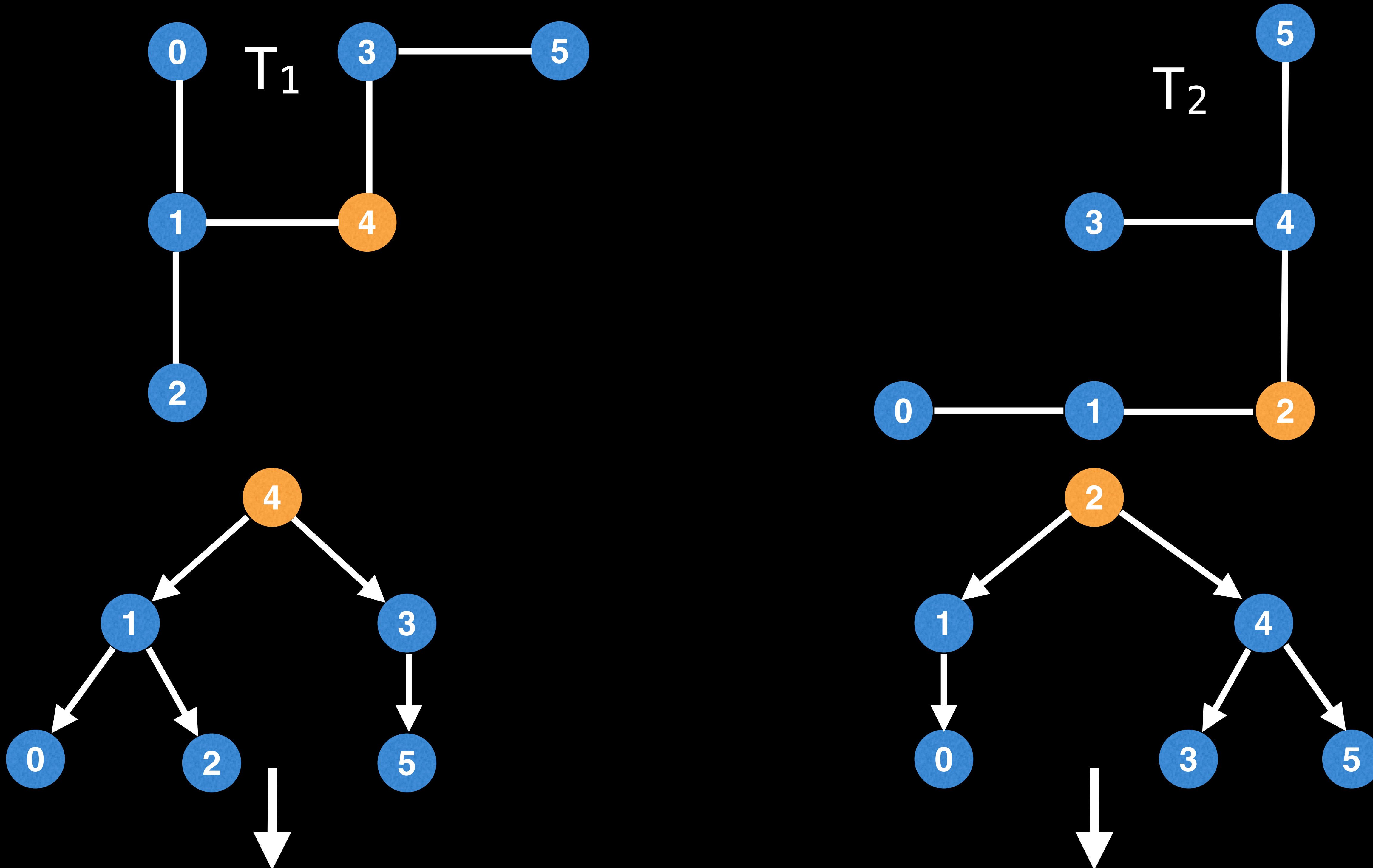
Find the center(s) of the original tree. We'll see how to handle the case where either tree can have more than 1 center shortly.



Root the tree at the center node.

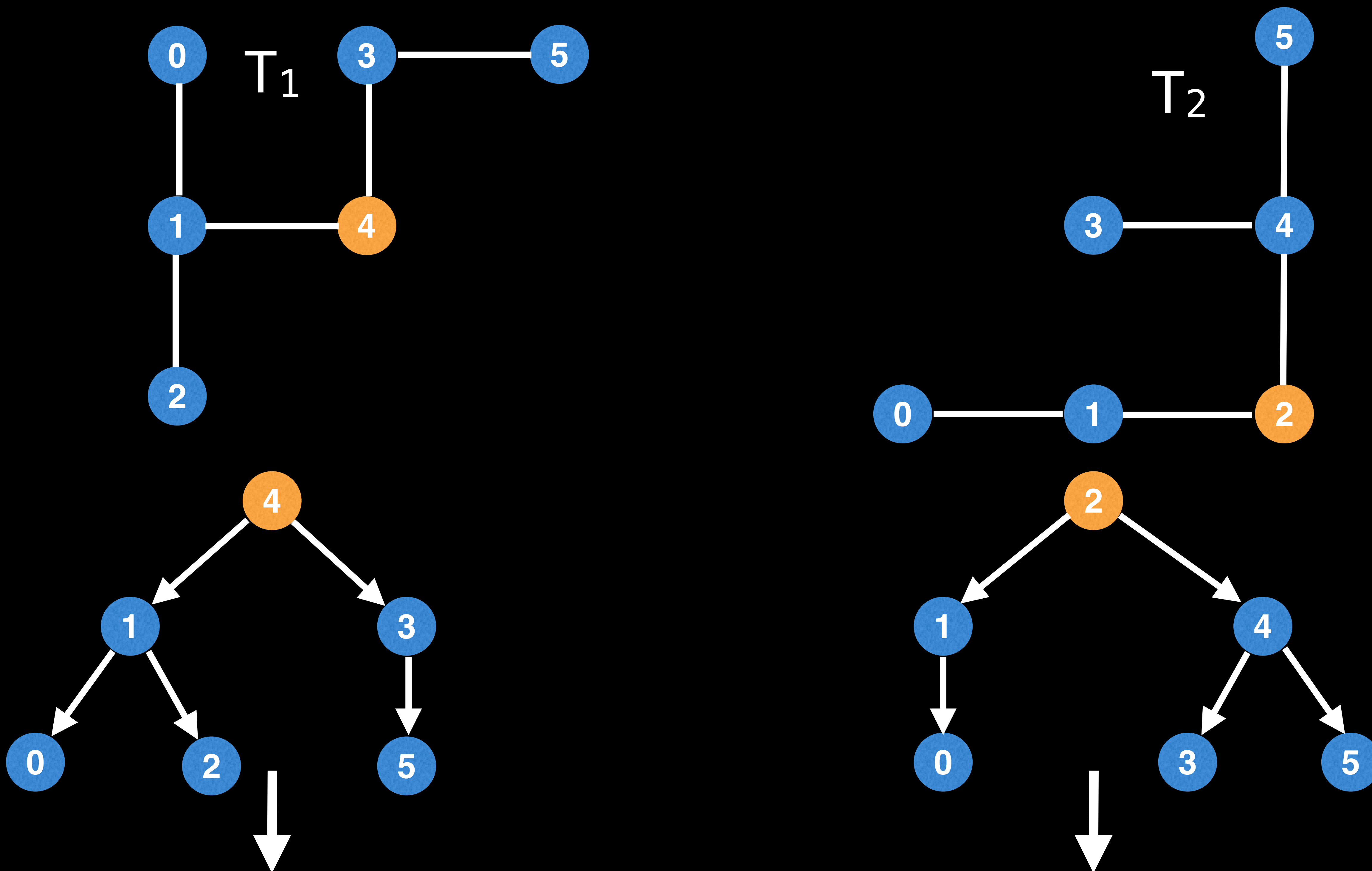


Generate the encoding for each tree and compare the serialized trees for equality.



000101100111 000101100111

The tree encoding is simply a sequence of left '(' and right ')' brackets. However, you can also think of them as 1's and 0's (i.e a large number) if you prefer.



((()())(()))   ((()())(()))

It should also be possible to reconstruct the tree solely from the encoding. This is left as an exercise to the reader… 😛

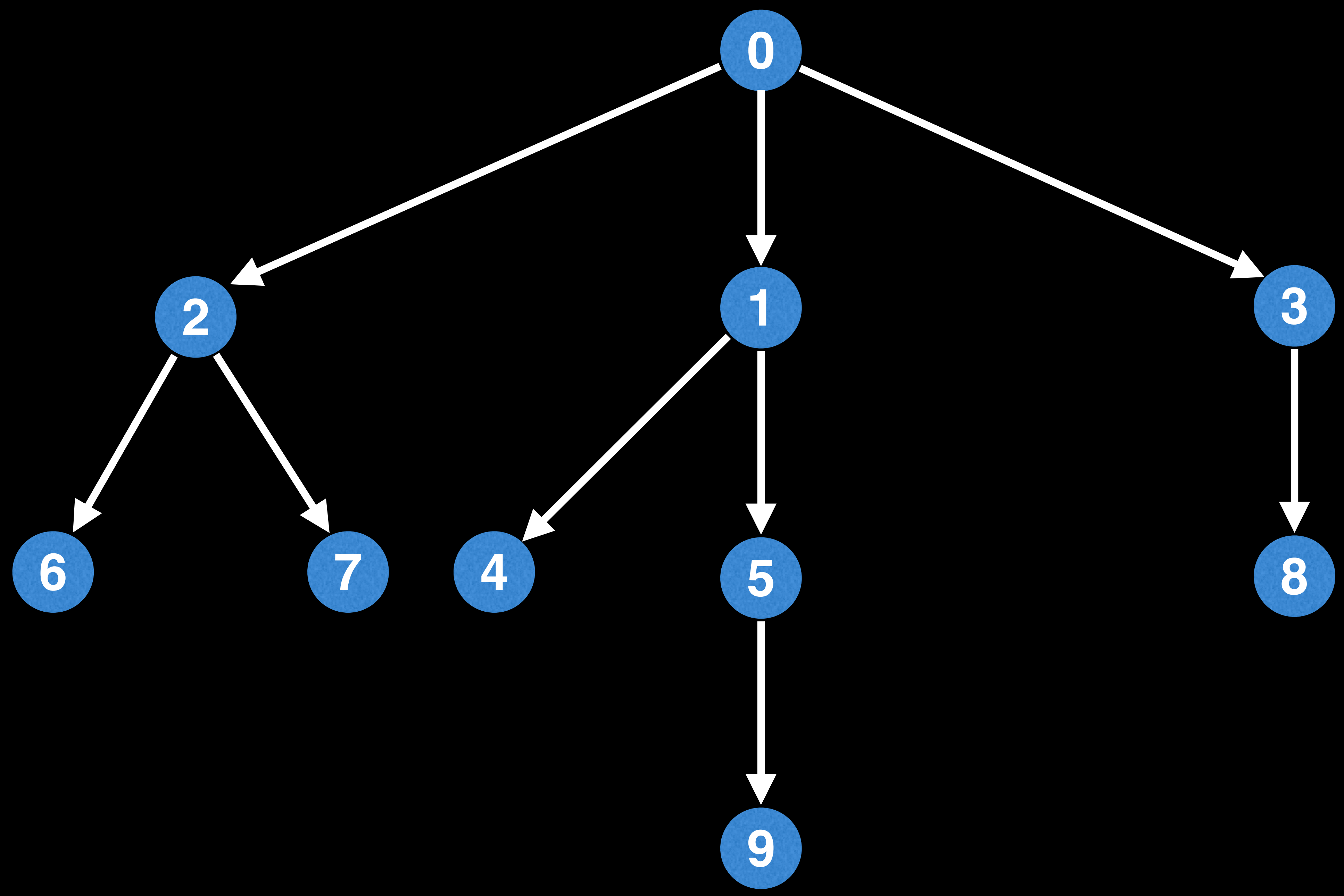
Generating the tree encoding

The AHU (Aho, Hopcroft, Ullman) algorithm is a clever serialization technique for representing a tree as a unique string.

Unlike many tree isomorphism invariants and heuristics, AHU is able to capture a **complete history** of a tree’s **degree spectrum** and structure ensuring a deterministic method of checking for tree isomorphisms.

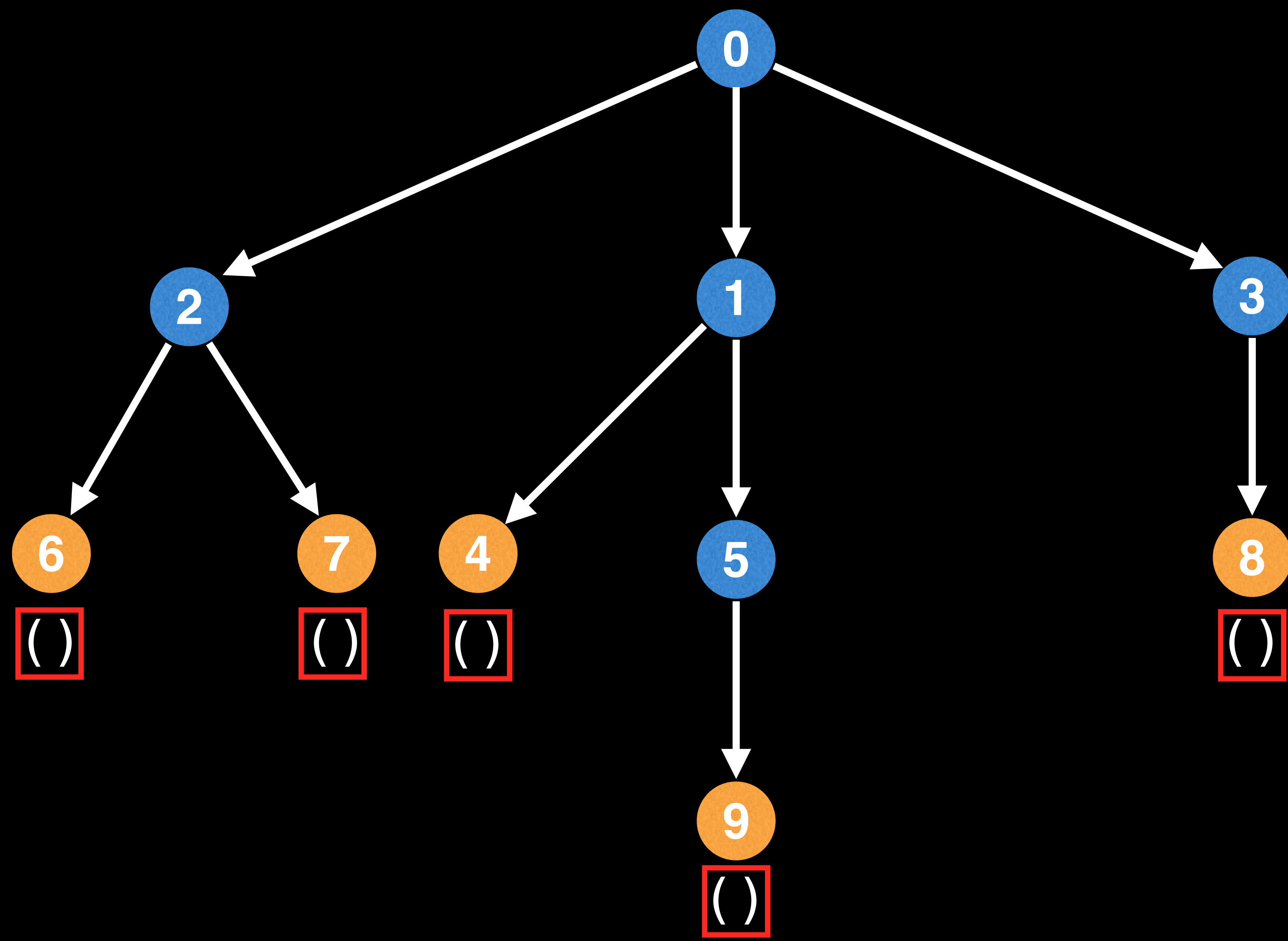
Let’s have a closer look…

Tree Encoding



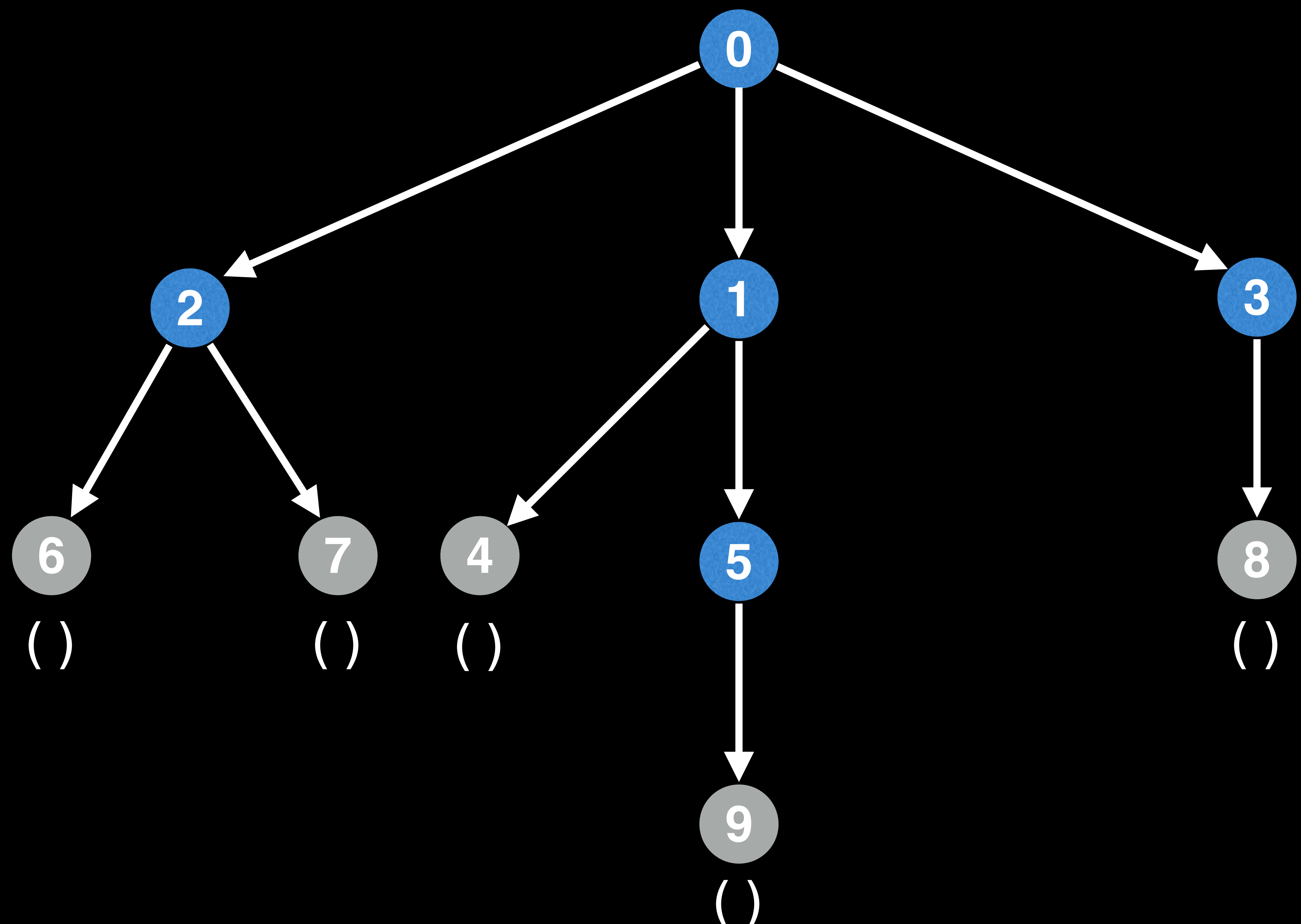
Tree Encoding

Start by assigning all leaf nodes Knuth tuples: '()'



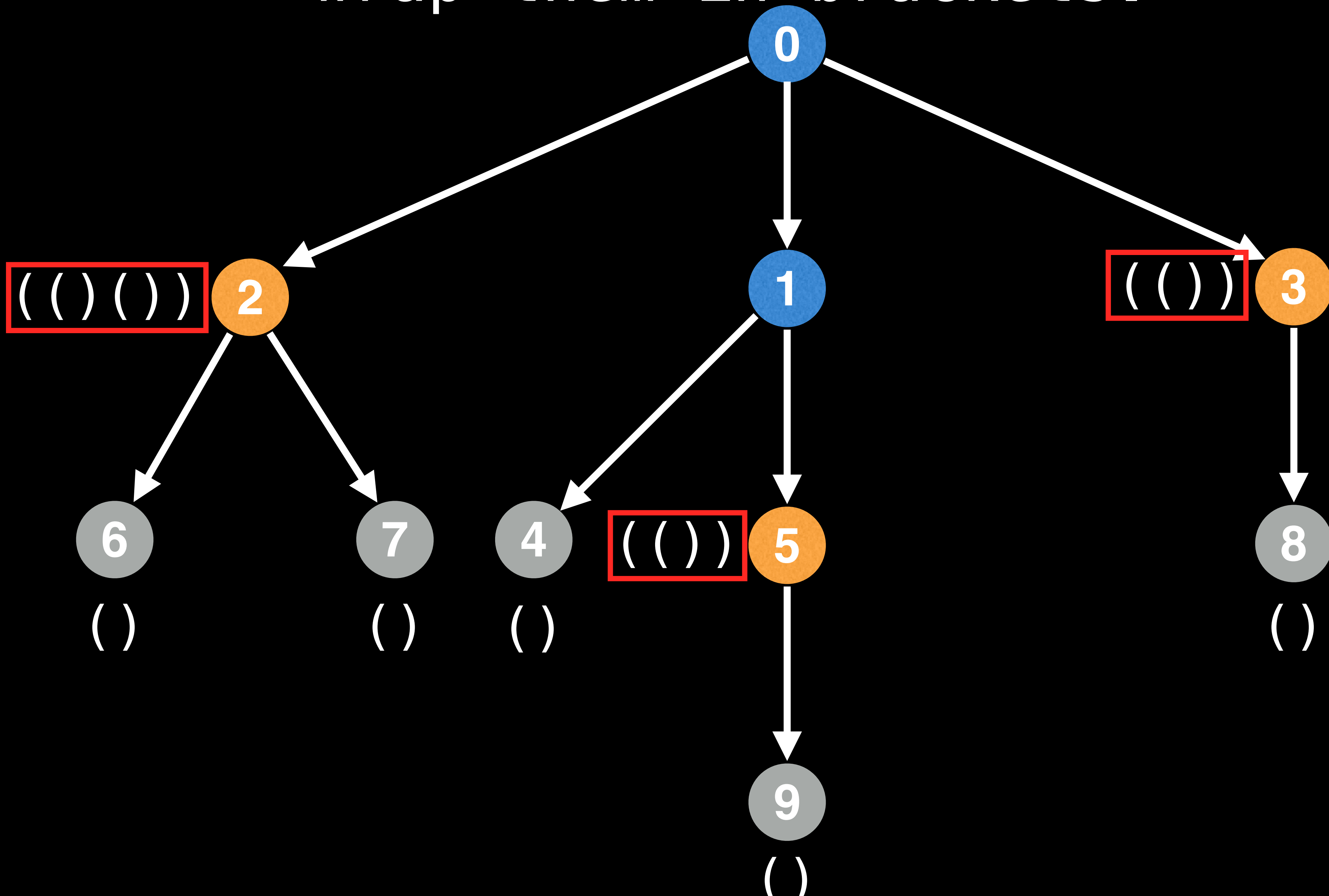
Tree Encoding

Start by assigning all leaf nodes Knuth tuples: '()'



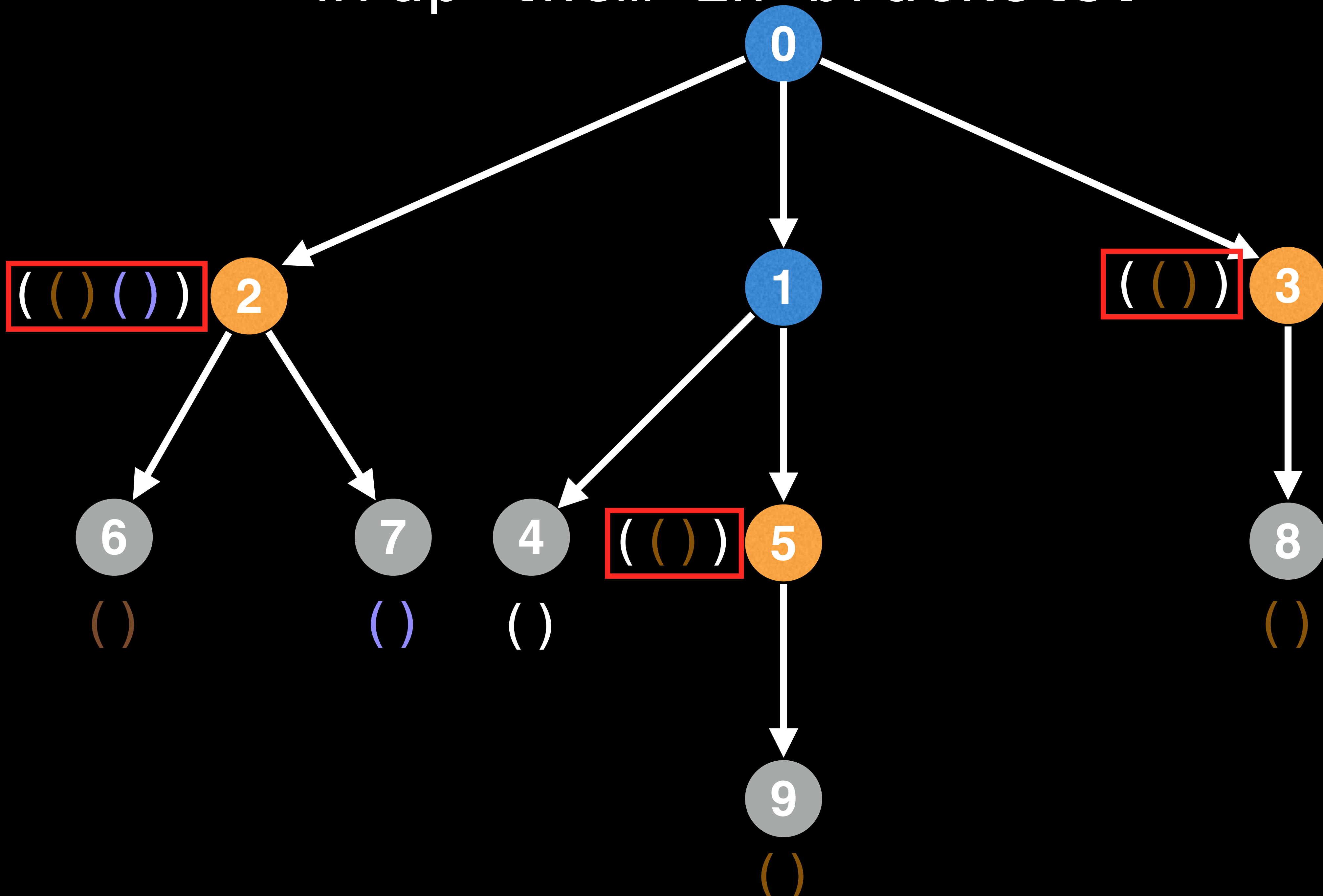
Tree Encoding

Process all nodes with grayed out children and combine the labels of their child nodes and wrap them in brackets.

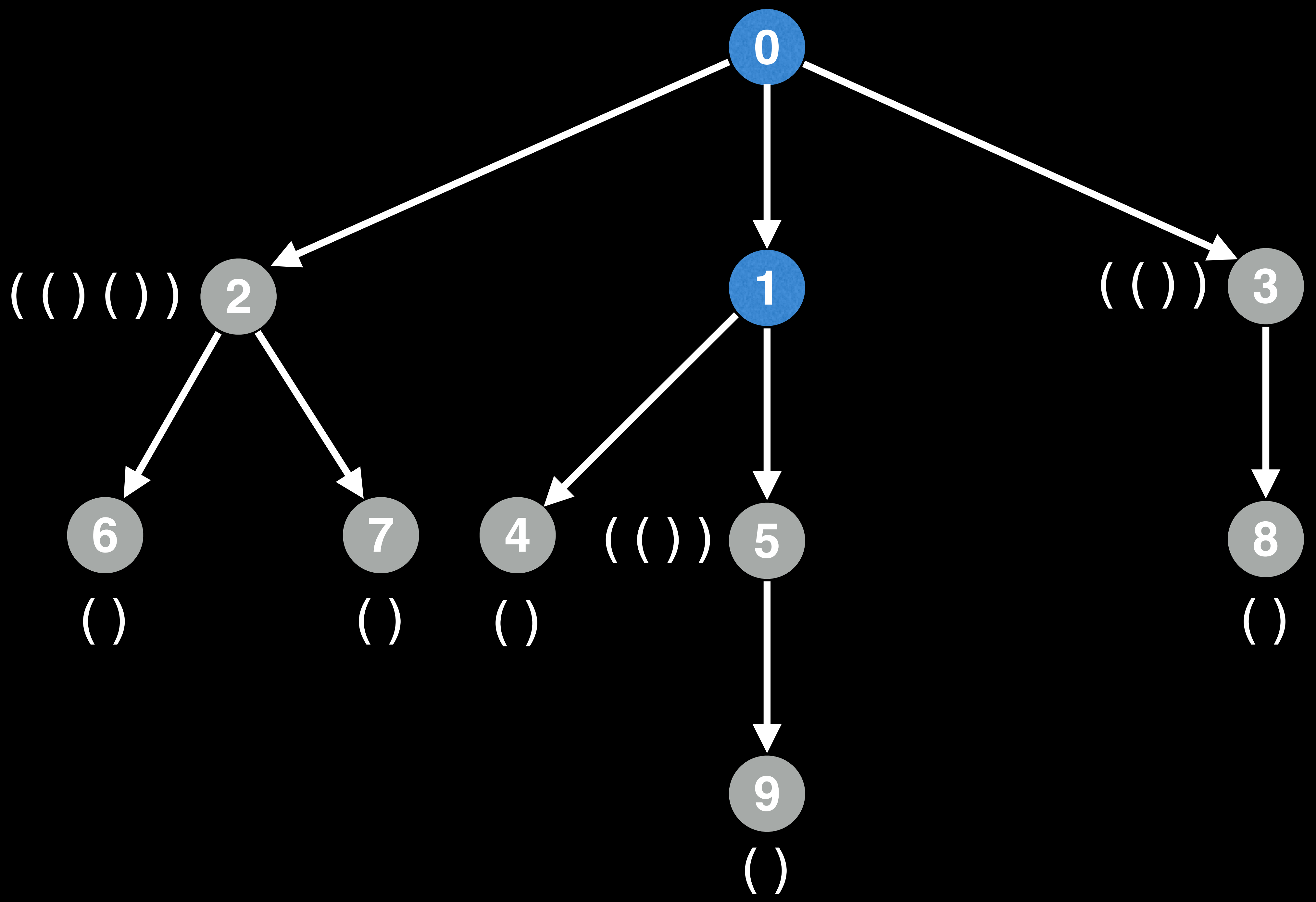


Tree Encoding

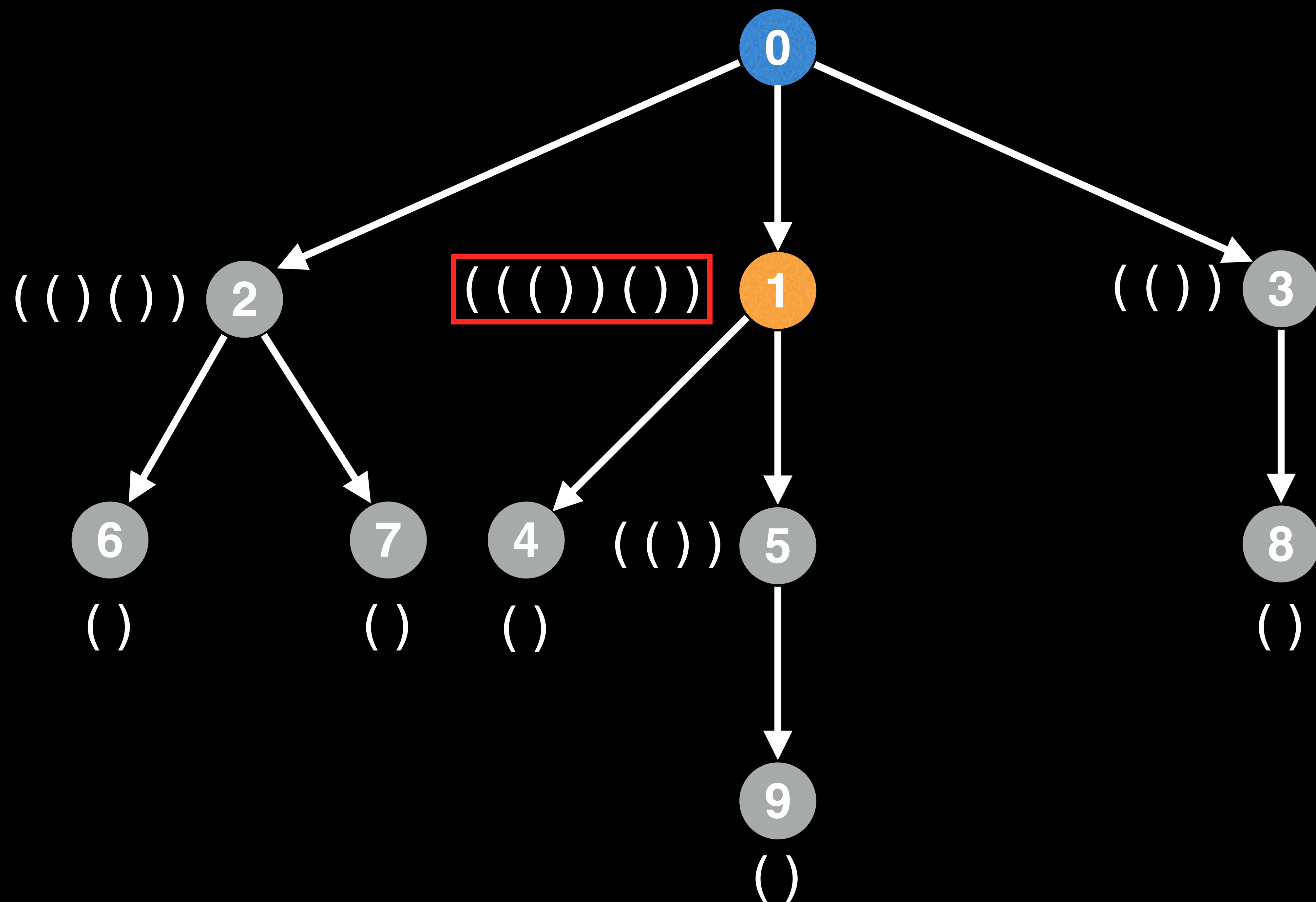
Process all nodes with grayed out children and combine the labels of their child nodes and wrap them in brackets.



Tree Encoding

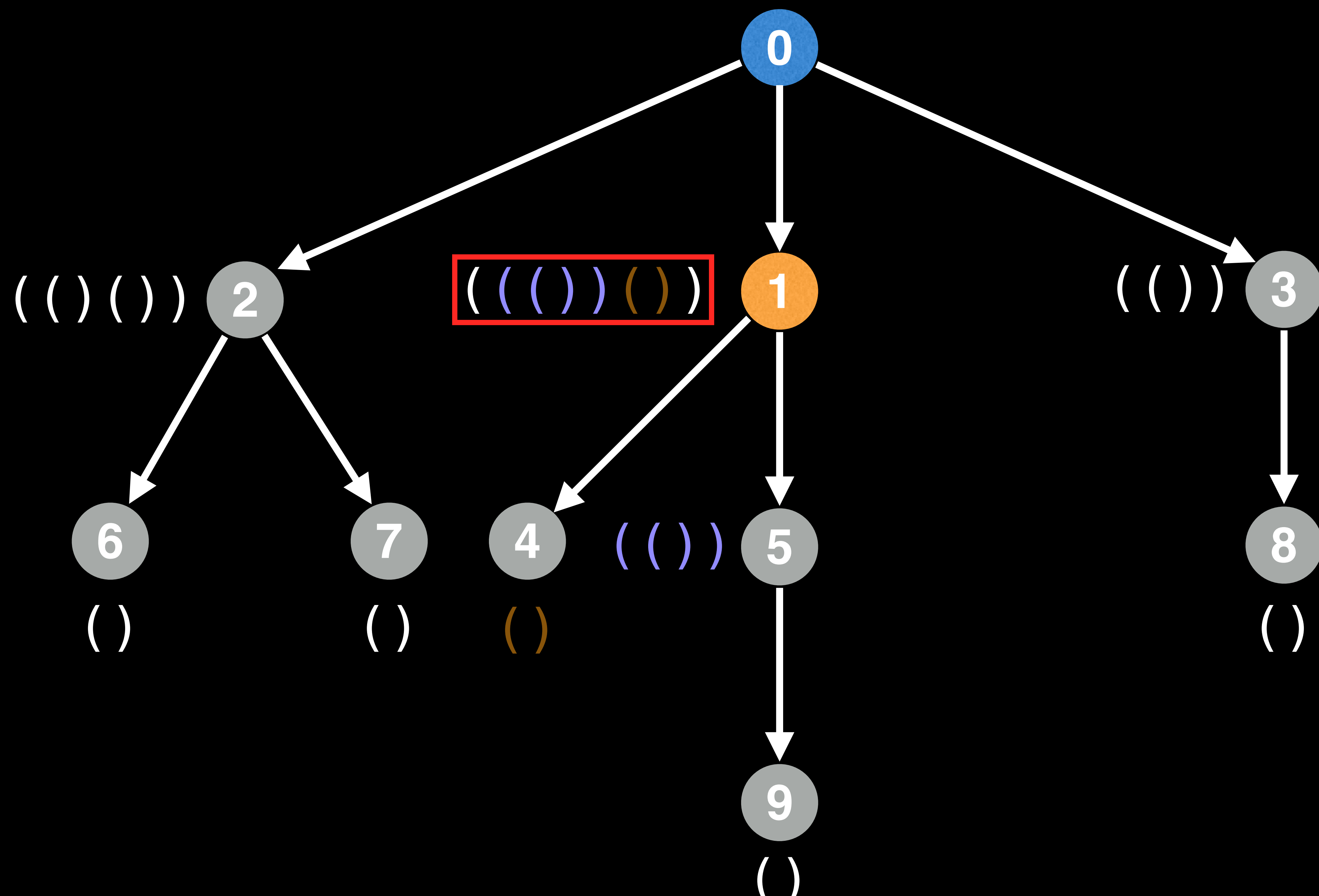


Tree Encoding



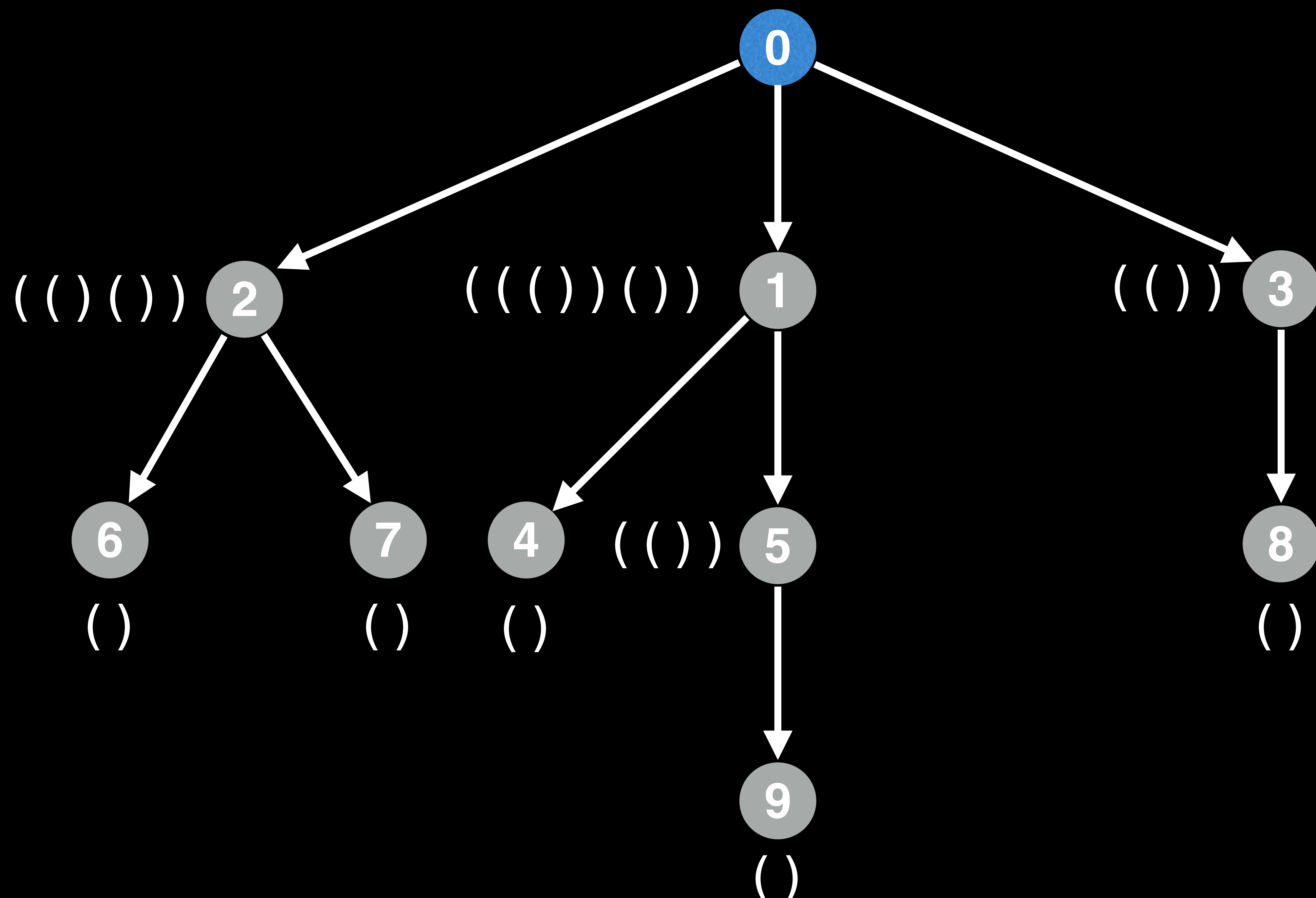
Tree Encoding

Notice that the labels get ***sorted*** when combined, this is important.

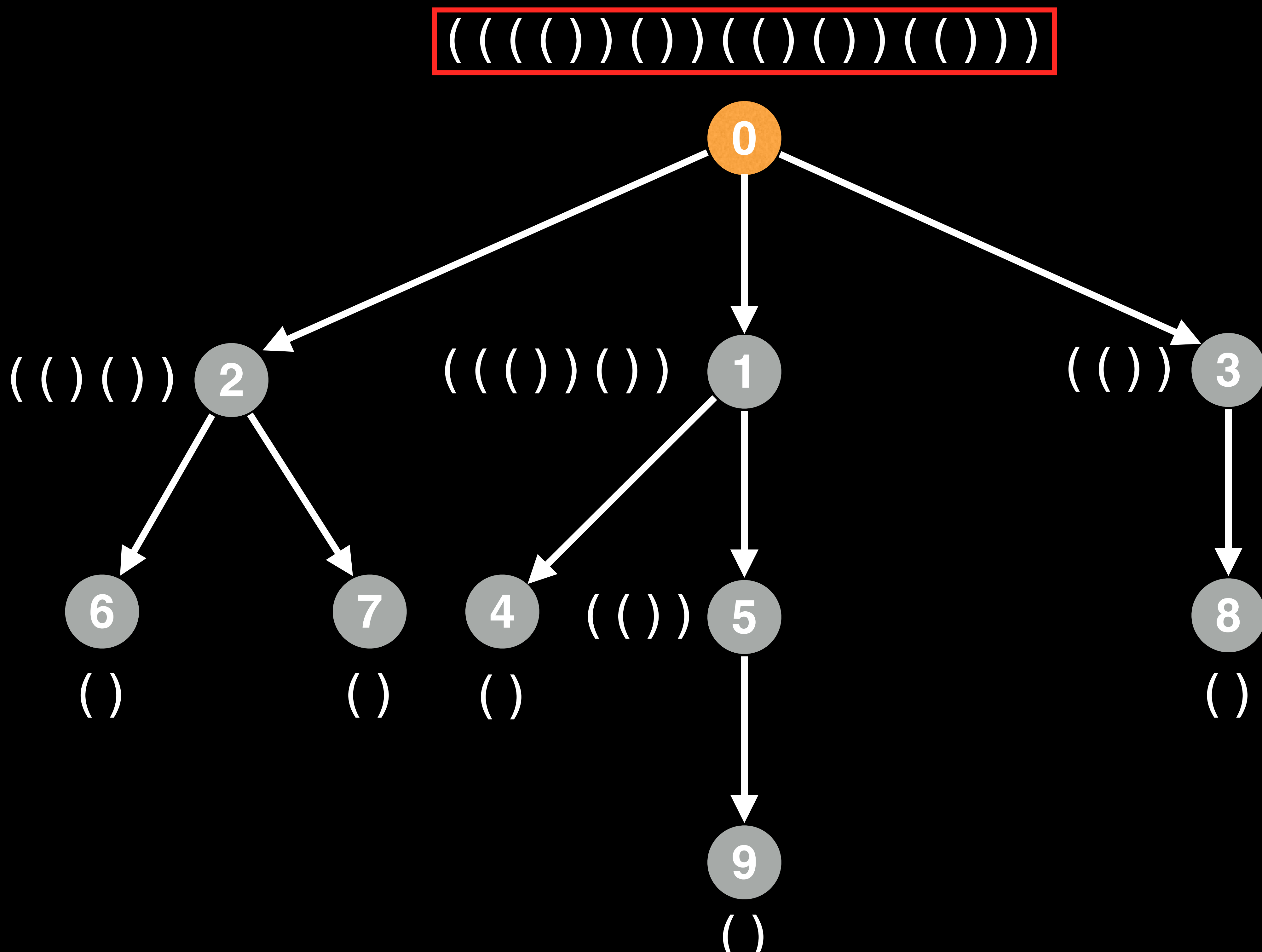


Tree Encoding

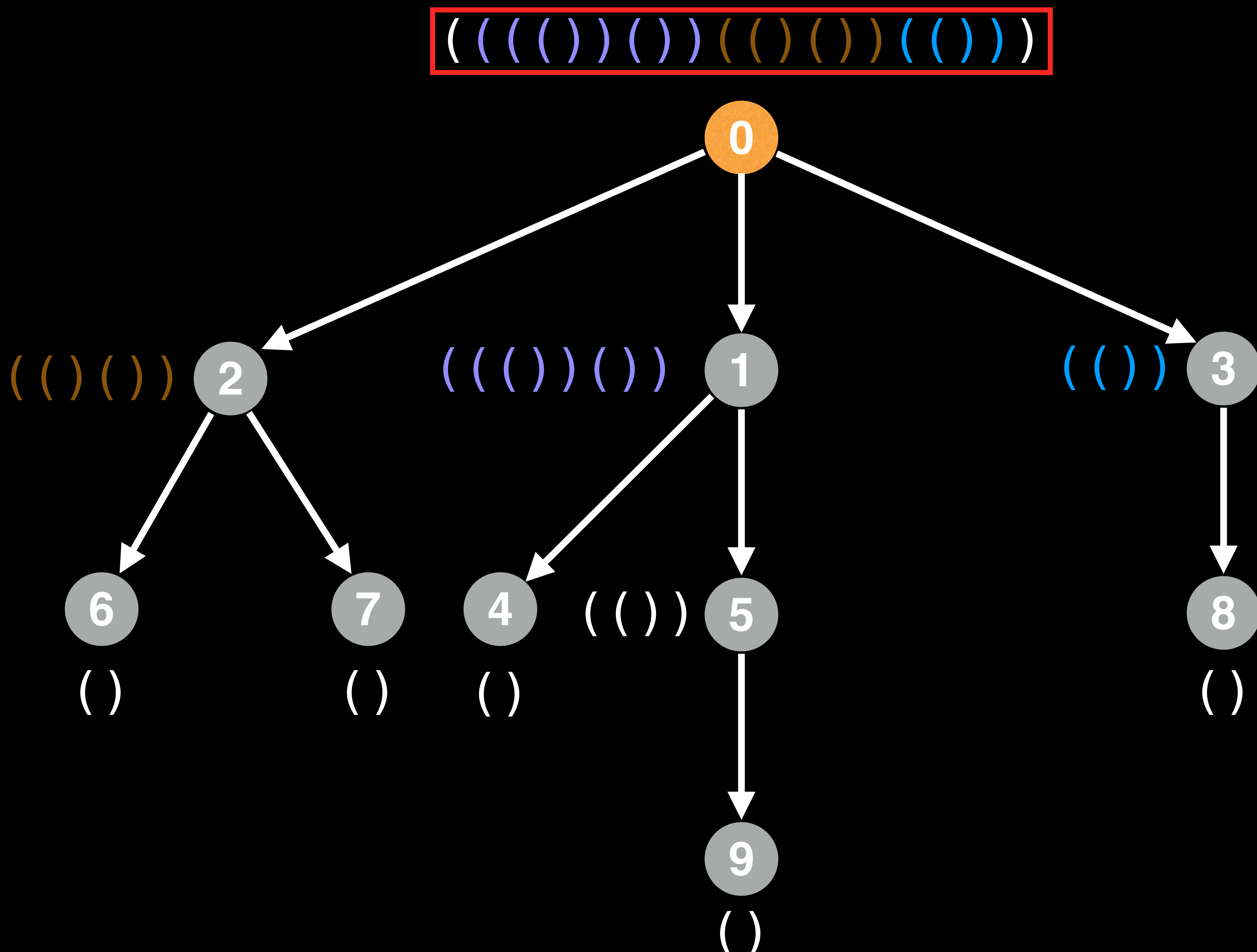
Notice that the labels get ***sorted*** when combined, this is important.



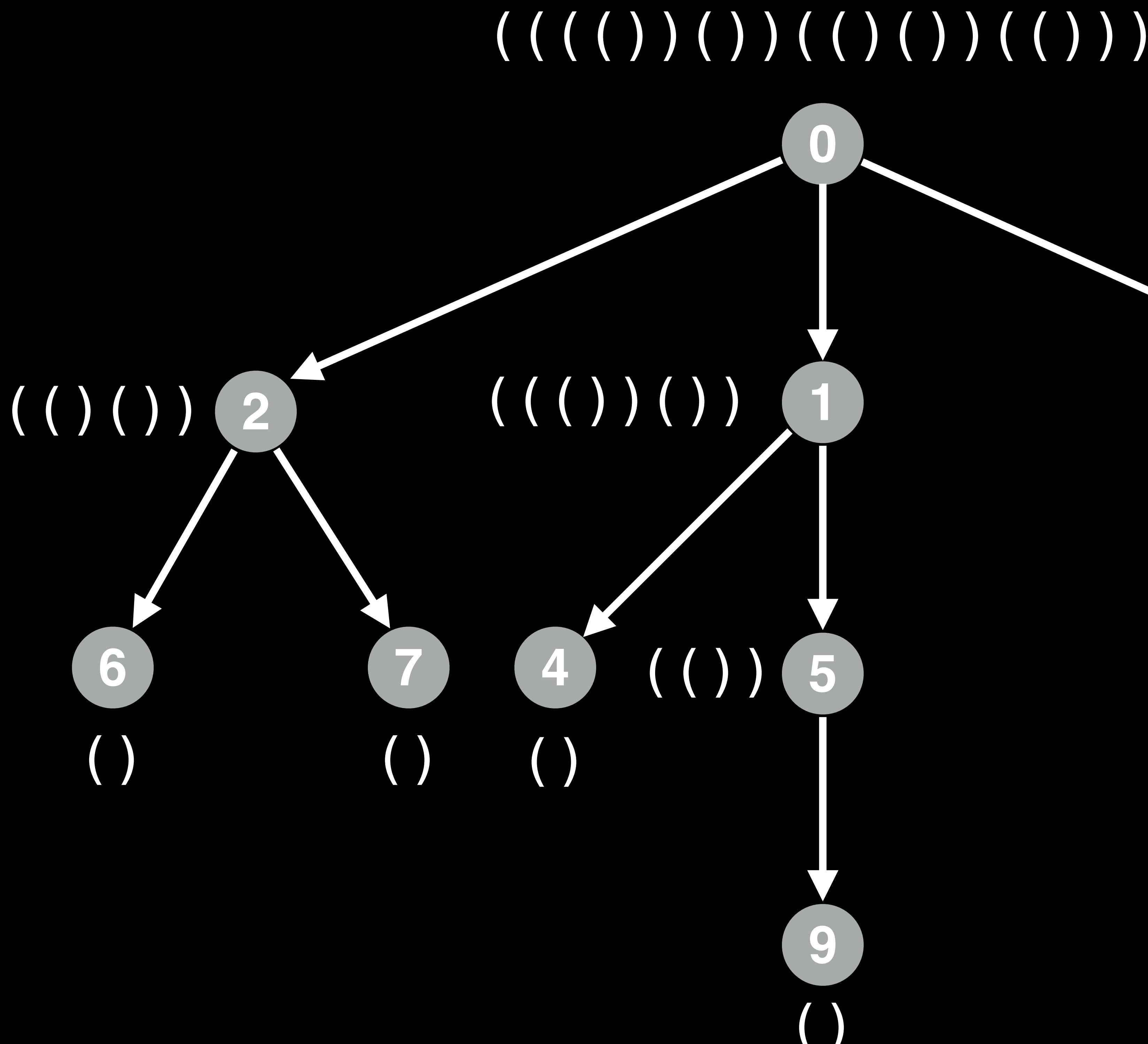
Tree Encoding



Tree Encoding



Tree Encoding



Tree Encoding Summary

In summary of what we did for AHU:

- Leaf nodes are assigned Knuth tuples '()' to begin with.
- Every time you move up a layer the labels of the previous subtrees get sorted lexicographically and wrapped in brackets.
- You cannot process a node until you have processed all its children.

Unrooted tree encoding pseudocode

```
# Returns whether two trees are isomorphic.
# Parameters tree1 and tree2 are undirected trees
# stored as adjacency lists.
function treesAreIsomorphic(tree1, tree2):
  tree1_centers = treeCenters(tree1)
  tree2_centers = treeCenters(tree2)

  tree1_rooted = rootTree(tree1, tree1_centers[0])
  tree1_encoded = encode(tree1_rooted)

  for center in tree2_centers:
    tree2_rooted = rootTree(tree2, center)
    tree2_encoded = encode(tree2_rooted)
    # Two trees are isomorphic if their encoded
    # canonical forms are equal.
    if tree1_encoded == tree2_encoded:
      return True
  return False
```

Unrooted tree encoding pseudocode

```
# Returns whether two trees are isomorphic.
# Parameters tree1 and tree2 are undirected trees
# stored as adjacency lists.
function treesAreIsomorphic(tree1, tree2):
  tree1_centers = treeCenters(tree1)
  tree2_centers = treeCenters(tree2)

  tree1_rooted = rootTree(tree1, tree1_centers[0])
  tree1_encoded = encode(tree1_rooted)

  for center in tree2_centers:
    tree2_rooted = rootTree(tree2, center)
    tree2_encoded = encode(tree2_rooted)
    # Two trees are isomorphic if their encoded
    # canonical forms are equal.
    if tree1_encoded == tree2_encoded:
      return True
  return False
```

Unrooted tree encoding pseudocode

```
# Returns whether two trees are isomorphic.
# Parameters tree1 and tree2 are undirected trees
# stored as adjacency lists.
function treesAreIsomorphic(tree1, tree2):
  tree1_centers = treeCenters(tree1)
  tree2_centers = treeCenters(tree2)

  tree1_rooted = rootTree(tree1, tree1_centers[0])
  tree1_encoded = encode(tree1_rooted)

  for center in tree2_centers:
    tree2_rooted = rootTree(tree2, center)
    tree2_encoded = encode(tree2_rooted)
    # Two trees are isomorphic if their encoded
    # canonical forms are equal.
    if tree1_encoded == tree2_encoded:
      return True
  return False
```

Unrooted tree encoding pseudocode

```
# Returns whether two trees are isomorphic.
# Parameters tree1 and tree2 are undirected trees
# stored as adjacency lists.
function treesAreIsomorphic(tree1, tree2):
  tree1_centers = treeCenters(tree1)
  tree2_centers = treeCenters(tree2)

  tree1_rooted = rootTree(tree1, tree1_centers[0])
  tree1_encoded = encode(tree1_rooted)

  for center in tree2_centers:
    tree2_rooted = rootTree(tree2, center)
    tree2_encoded = encode(tree2_rooted)
    # Two trees are isomorphic if their encoded
    # canonical forms are equal.
    if tree1_encoded == tree2_encoded:
      return True
  return False
```

Unrooted tree encoding pseudocode

Rooted trees are stored recursively in TreeNode objects:

```
# TreeNode object structure.
class TreeNode:
  # Unique integer id to identify this node.
  int id;

  # Pointer to parent TreeNode reference. Only the
  # root node has a null parent TreeNode reference.
  TreeNode parent;

  # List of pointers to child TreeNodes.
  TreeNode[] children;
```

Unrooted tree encoding pseudocode

```
# Returns whether two trees are isomorphic.
# Parameters tree1 and tree2 are undirected trees
# stored as adjacency lists.
function treesAreIsomorphic(tree1, tree2):
  tree1_centers = treeCenters(tree1)
  tree2_centers = treeCenters(tree2)

  tree1_rooted = rootTree(tree1, tree1_centers[0])
  tree1_encoded = encode(tree1_rooted)

  for center in tree2_centers:
    tree2_rooted = rootTree(tree2, center)
    tree2_encoded = encode(tree2_rooted)
    # Two trees are isomorphic if their encoded
    # canonical forms are equal.
    if tree1_encoded == tree2_encoded:
      return True
  return False
```

Unrooted tree encoding pseudocode

```
function encode(node):
  if node == null:
    return ""

  labels = []
  for child in node.children():
    labels.add(encode(child))

  # Regular lexicographic sort
  sort(labels)

  result = ""
  for label in labels:
    result += label

  return "(" + result + ")"
```

Unrooted tree encoding pseudocode

```
function encode(node):
  if node == null:
    return ""

  labels = []
  for child in node.children():
    labels.add(encode(child))

  # Regular lexicographic sort
  sort(labels)

  result = ""
  for label in labels:
    result += label

  return "(" + result + ")"
```

Unrooted tree encoding pseudocode

```
function encode(node):
  if node == null:
    return ""

  labels = []
  for child in node.children():
    labels.add(encode(child))

  # Regular lexicographic sort
  sort(labels)

  result = ""
  for label in labels:
    result += label

  return "(" + result + ")"
```

Unrooted tree encoding pseudocode

```
function encode(node):
  if node == null:
    return ""

  labels = []
  for child in node.children():
    labels.add(encode(child))

  # Regular lexicographic sort
  sort(labels)

  result = ""
  for label in labels:
    result += label

  return "(" + result + ")"
```

Unrooted tree encoding pseudocode

```
function encode(node):
  if node == null:
    return ""

  labels = []
  for child in node.children():
    labels.add(encode(child))

  # Regular lexicographic sort
  sort(labels)

  result = ""
  for label in labels:
    result += label

  return "(" + result + ")"
```

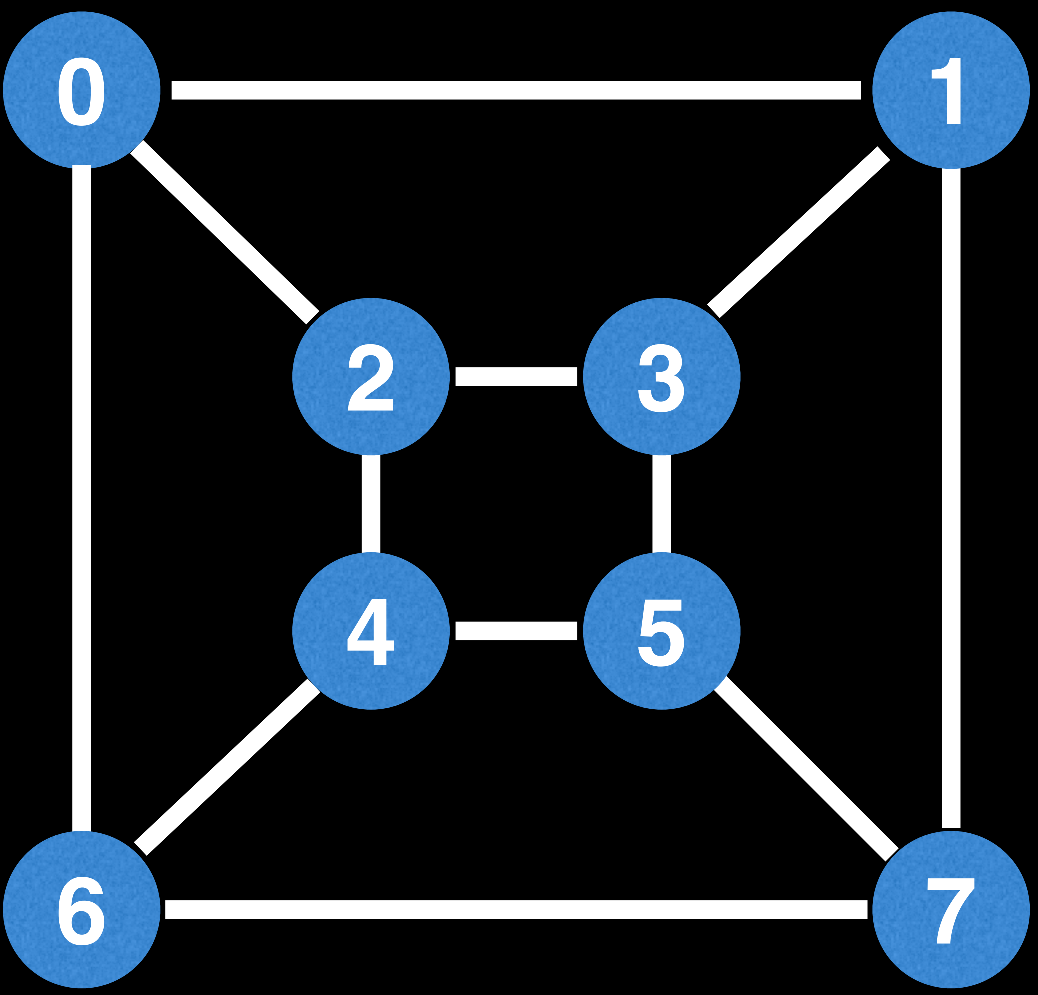
Unrooted tree encoding pseudocode

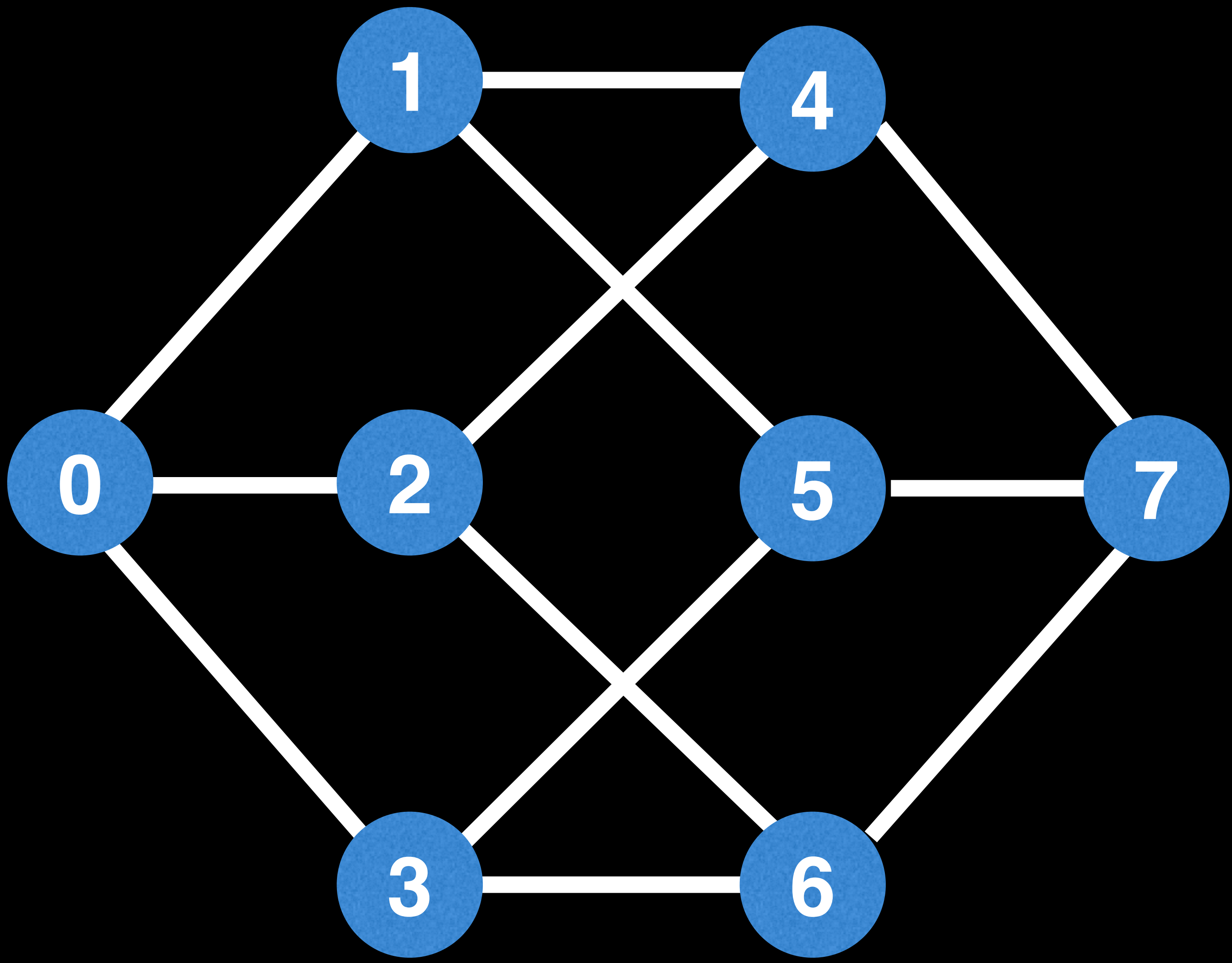
```
# Returns whether two trees are isomorphic.
# Parameters tree1 and tree2 are undirected trees
# stored as adjacency lists.
function treesAreIsomorphic(tree1, tree2):
  tree1_centers = treeCenters(tree1)
  tree2_centers = treeCenters(tree2)

  tree1_rooted = rootTree(tree1, tree1_centers[0])
  tree1_encoded = encode(tree1_rooted)

  for center in tree2_centers:
    tree2_rooted = rootTree(tree2, center)
    tree2_encoded = encode(tree2_rooted)
    # Two trees are isomorphic if their encoded
    # canonical forms are equal.
    if tree1_encoded == tree2_encoded:
      return True
  return False
```

Unrooted tree encoding pseudocode

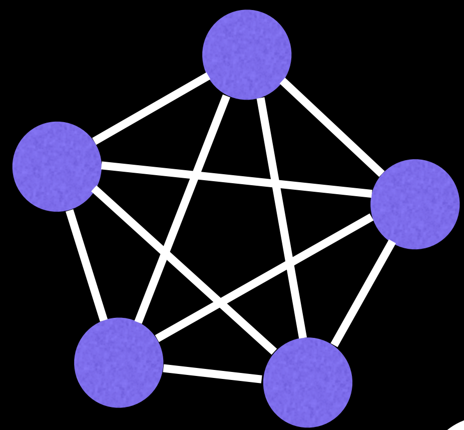
```
# Returns whether two trees are isomorphic.
# Parameters tree1 and tree2 are undirected trees
# stored as adjacency lists.
function treesAreIsomorphic(tree1, tree2):
  tree1_centers = treeCenters(tree1)
  tree2_centers = treeCenters(tree2)

  tree1_rooted = rootTree(tree1, tree1_centers[0])
  tree1_encoded = encode(tree1_rooted)

  for center in tree2_centers:
    tree2_rooted = rootTree(tree2, center)
    tree2_encoded = encode(tree2_rooted)
    # Two trees are isomorphic if their encoded
    # canonical forms are equal.
    if tree1_encoded == tree2_encoded:
      return True
  return False
```

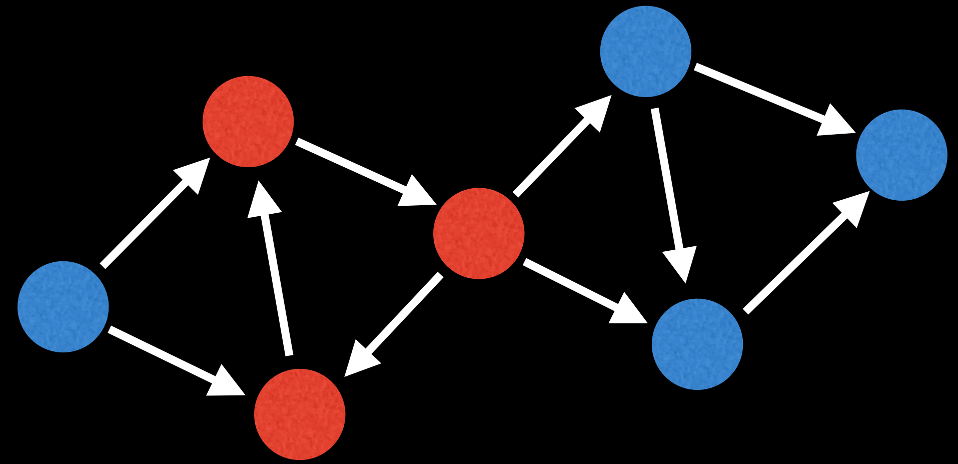
Isomorphic Trees

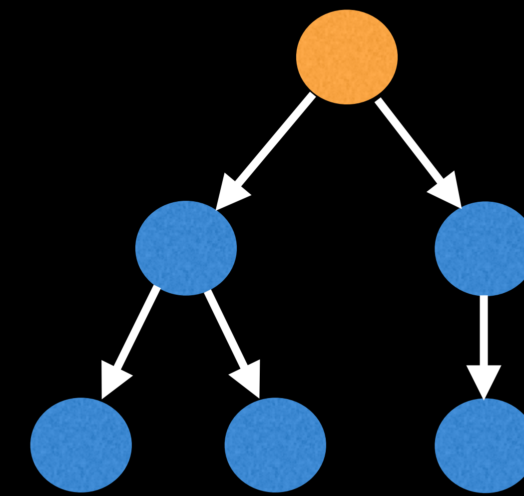




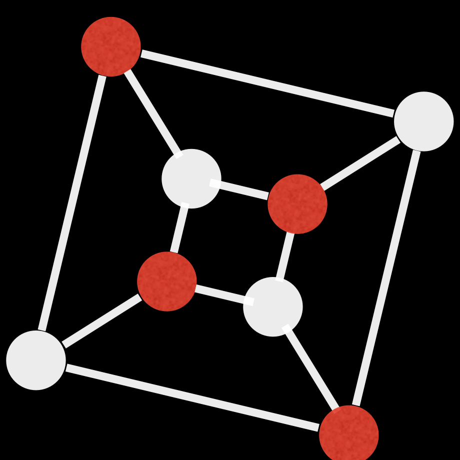
Determining if two graphs are isomorphic is not only not obvious to the human eye, but also a difficult problem for computers.

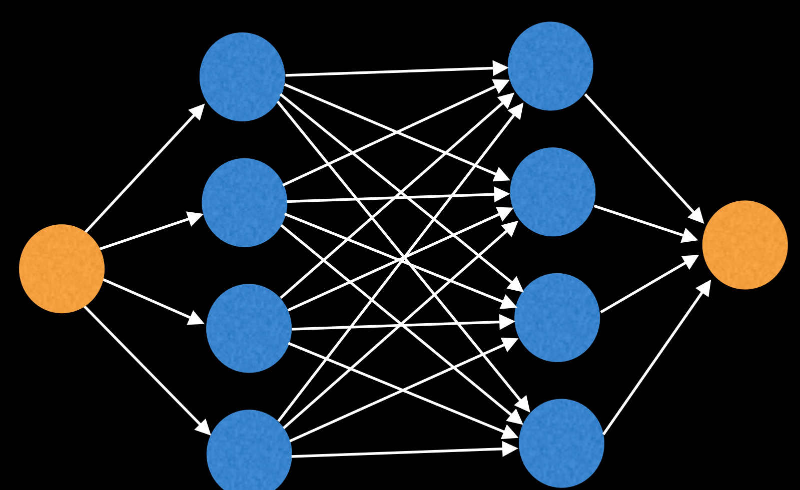


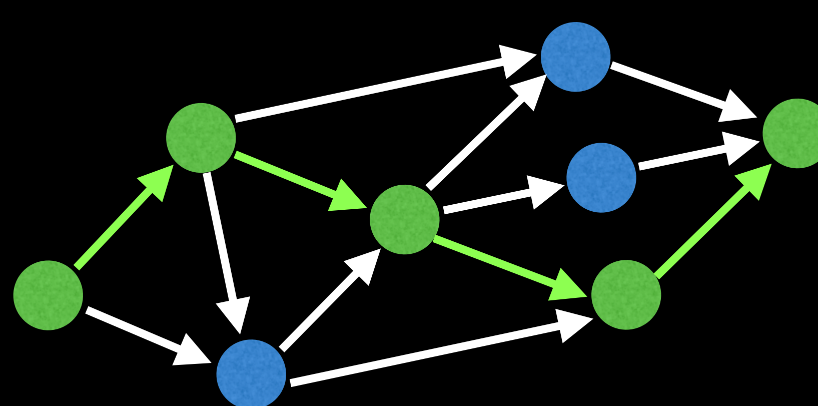




Graph Theory Video Series







Isomorphisms in trees source code

A question of equality

William Fiset

Previous video explaining identifying isomorphic trees:

Source Code Link

Implementation source code can be found at the following link:

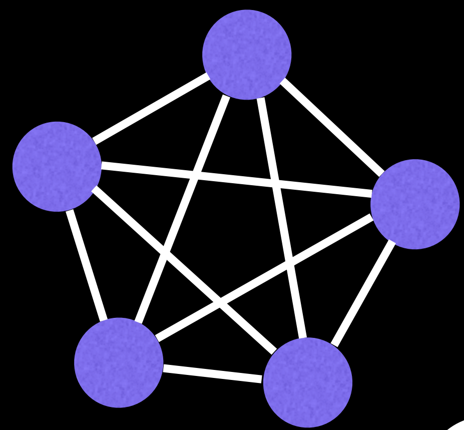
github.com/williamfiset/algorithms

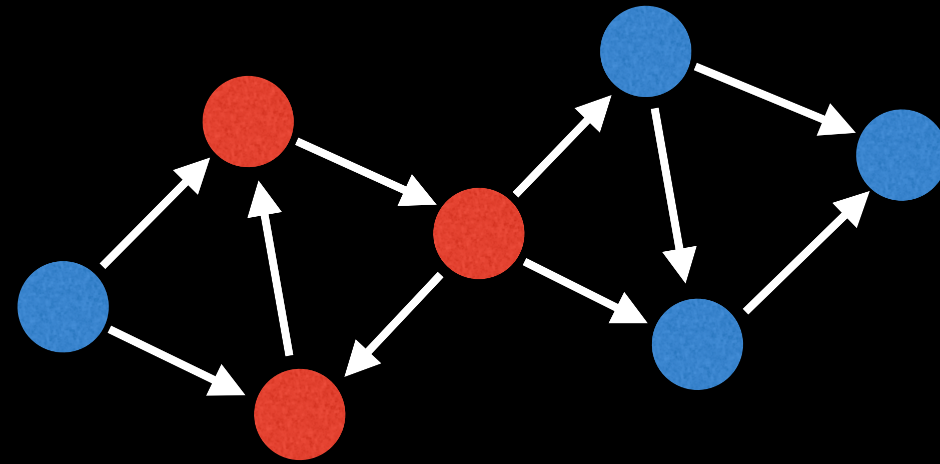
Link in the description below:

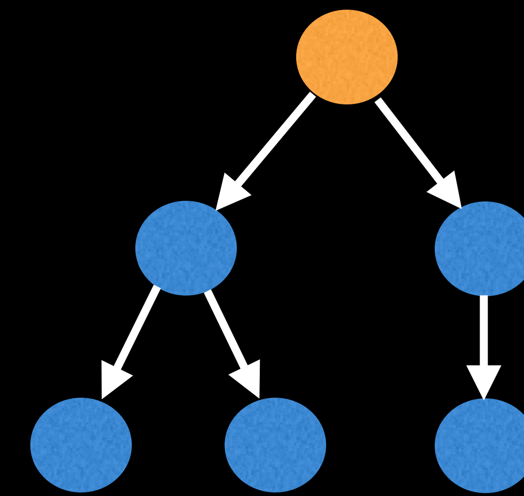
Isomorphic Trees Source Code

A question of equality

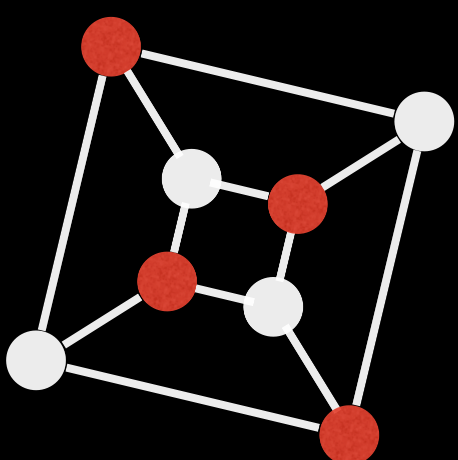
William Fiset

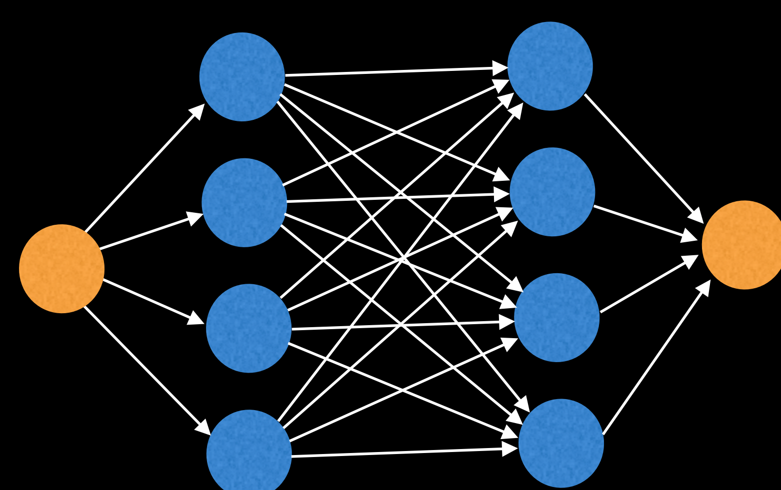


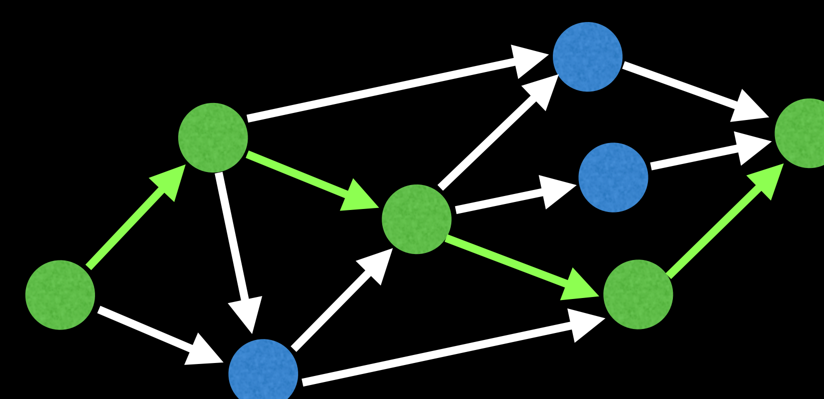




Graph Theory Video Series







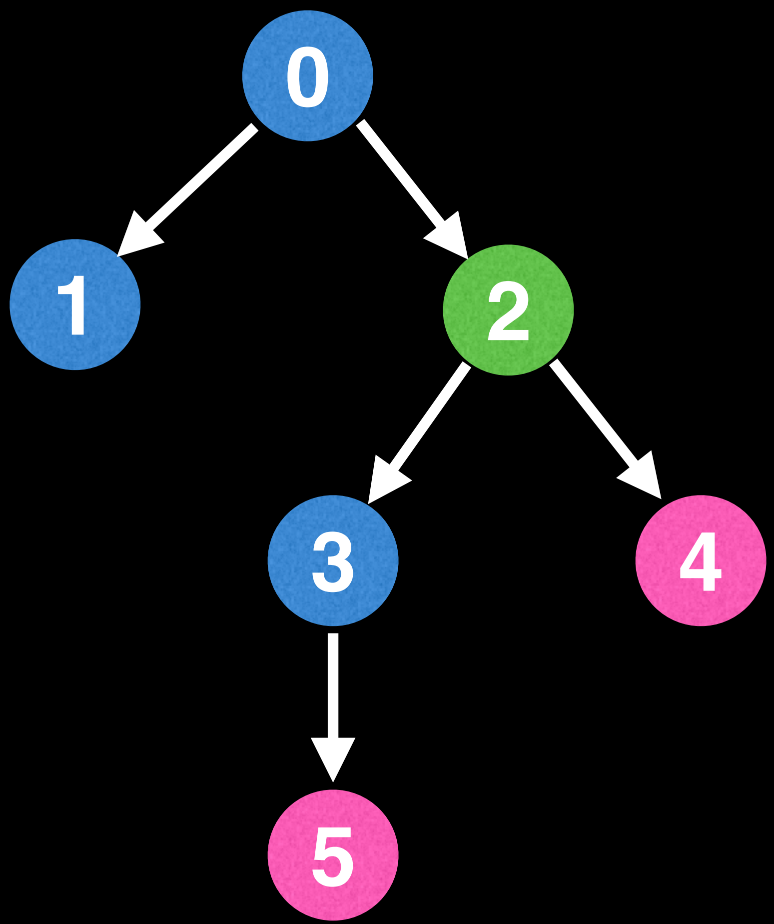
Lowest Common Ancestor

Eulerian tour + range minimum query method

William Fiset

Definition

The **Lowest Common Ancestor** (LCA) of two nodes `a` and `b` in a **rooted tree** is the deepest node `c` that has both `a` and `b` as descendants (where a node can be a descendant of itself)

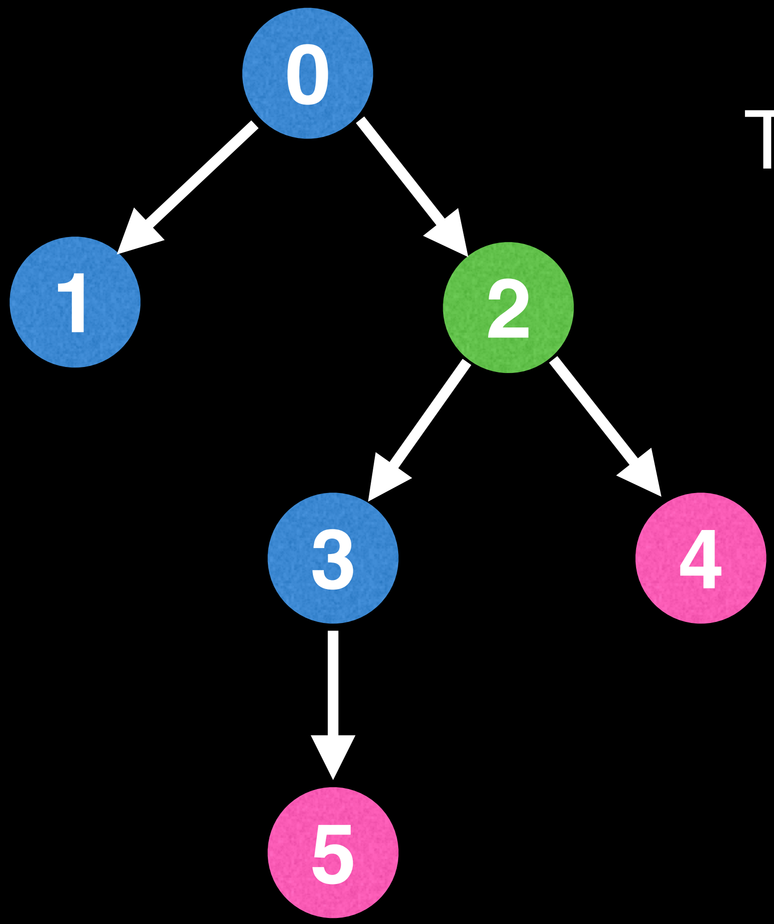


LCA(5, 4) = 2

NOTE: The notion of a LCA also exists for Directed Acyclic Graphs (DAGs), but today we’re only looking at the LCA in the context of trees.

Definition

The **Lowest Common Ancestor** (LCA) of two nodes \`a\` and \`b\` in a **rooted tree** is the deepest node \`c\` that has both \`a\` and \`b\` as descendants (where a node can be a descendant of itself)



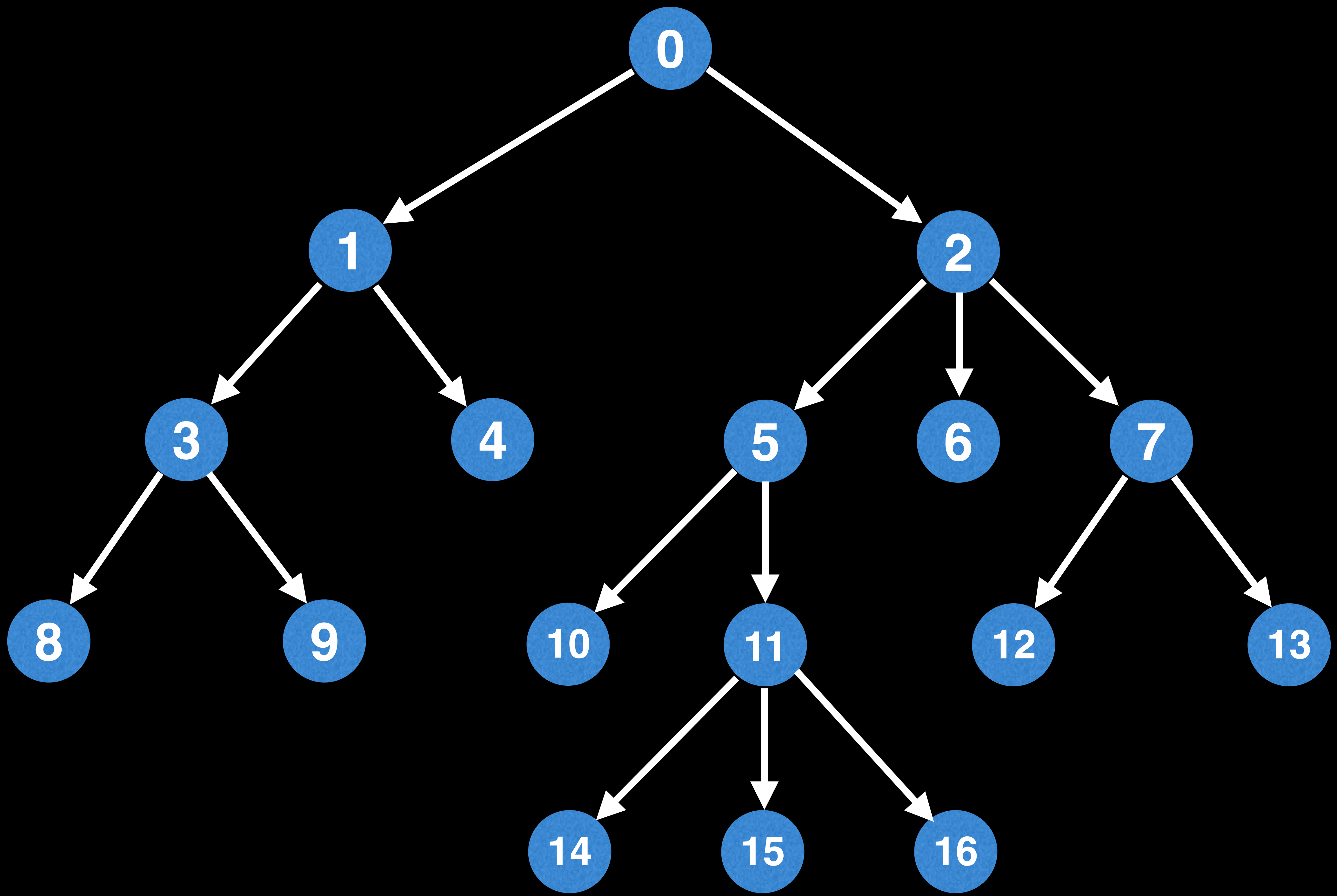
LCA(5, 4) = 2

The LCA problem has several applications in Computer Science, notably:

- Finding the distance between two nodes
- Inheritance hierarchies in OOP
- As a subroutine in several advanced algorithms and data structures
- etc…

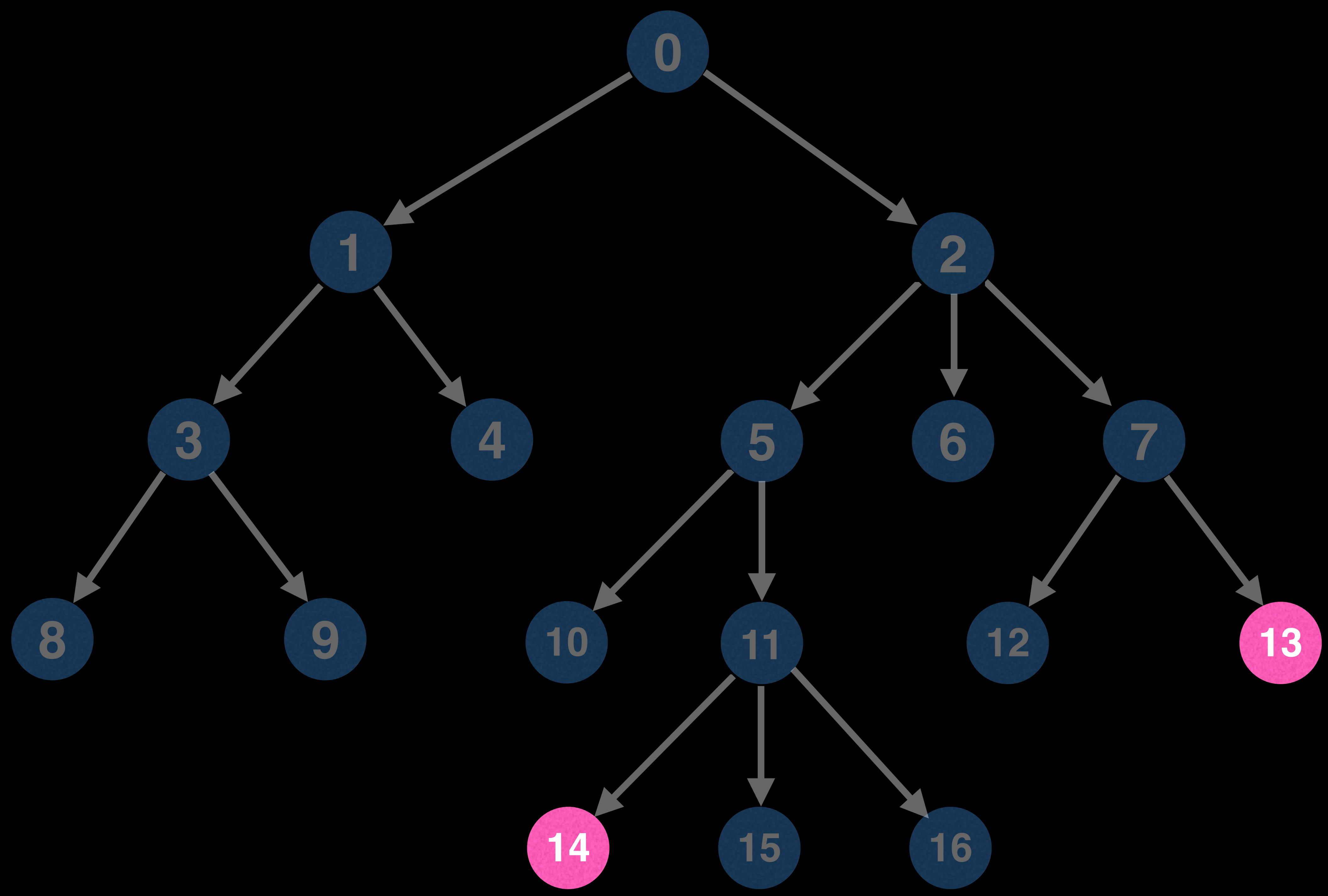
NOTE: The notion of a LCA also exists for Directed Acyclic Graphs (DAGs), but today we're only looking at the LCA in the context of trees.

Understanding LCA



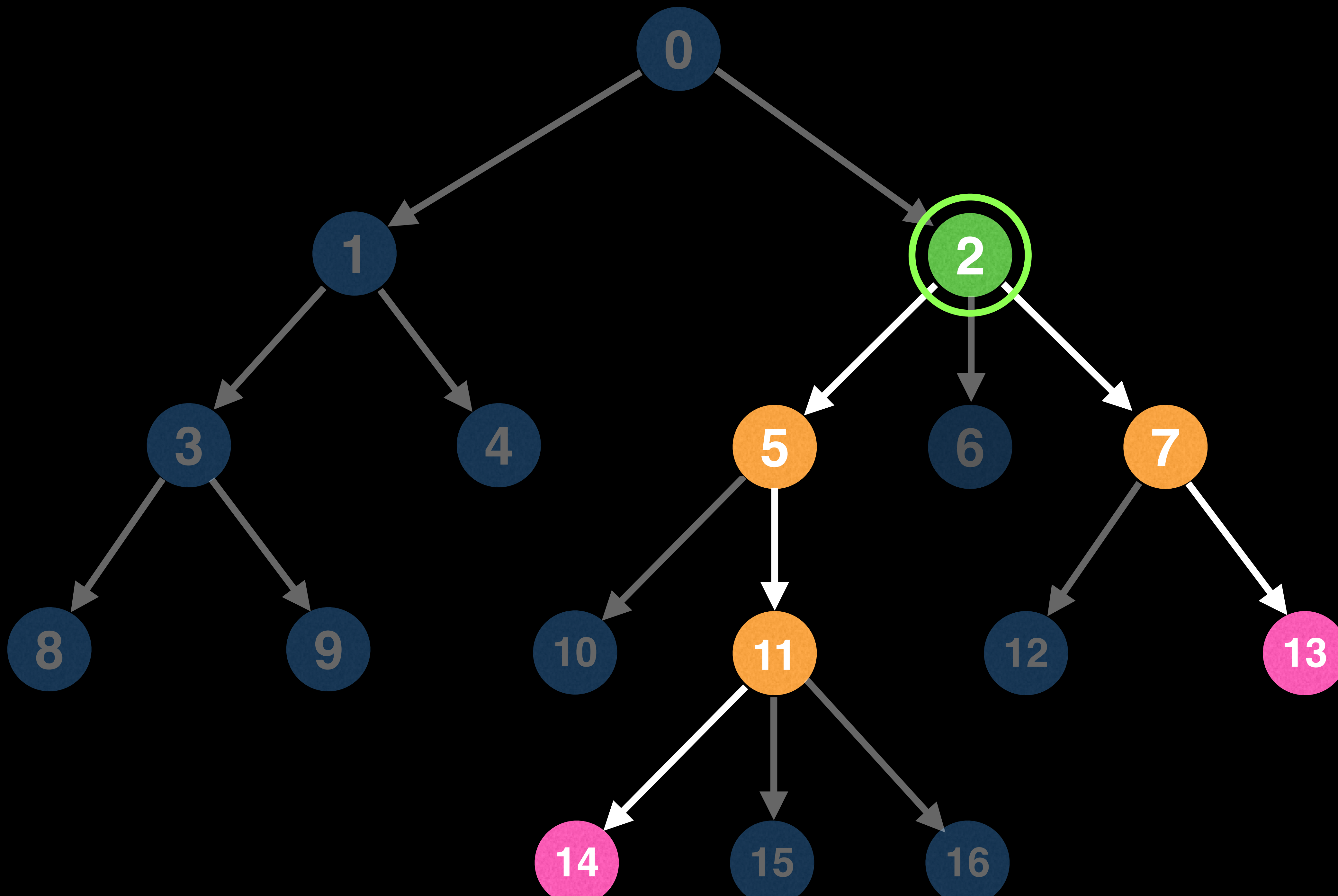
Understanding LCA

LCA(13, 14)



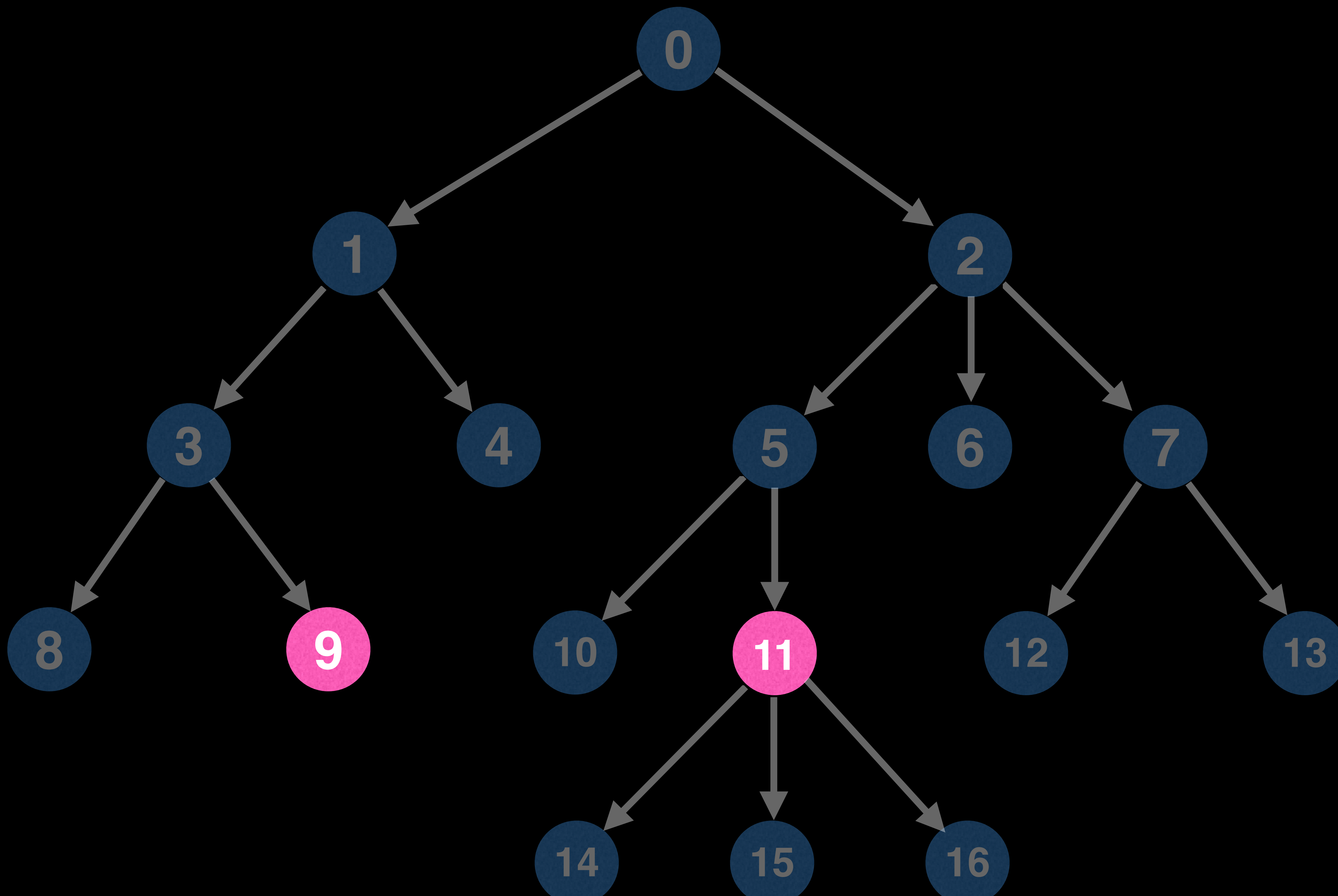
Understanding LCA

LCA(13, 14) = 2



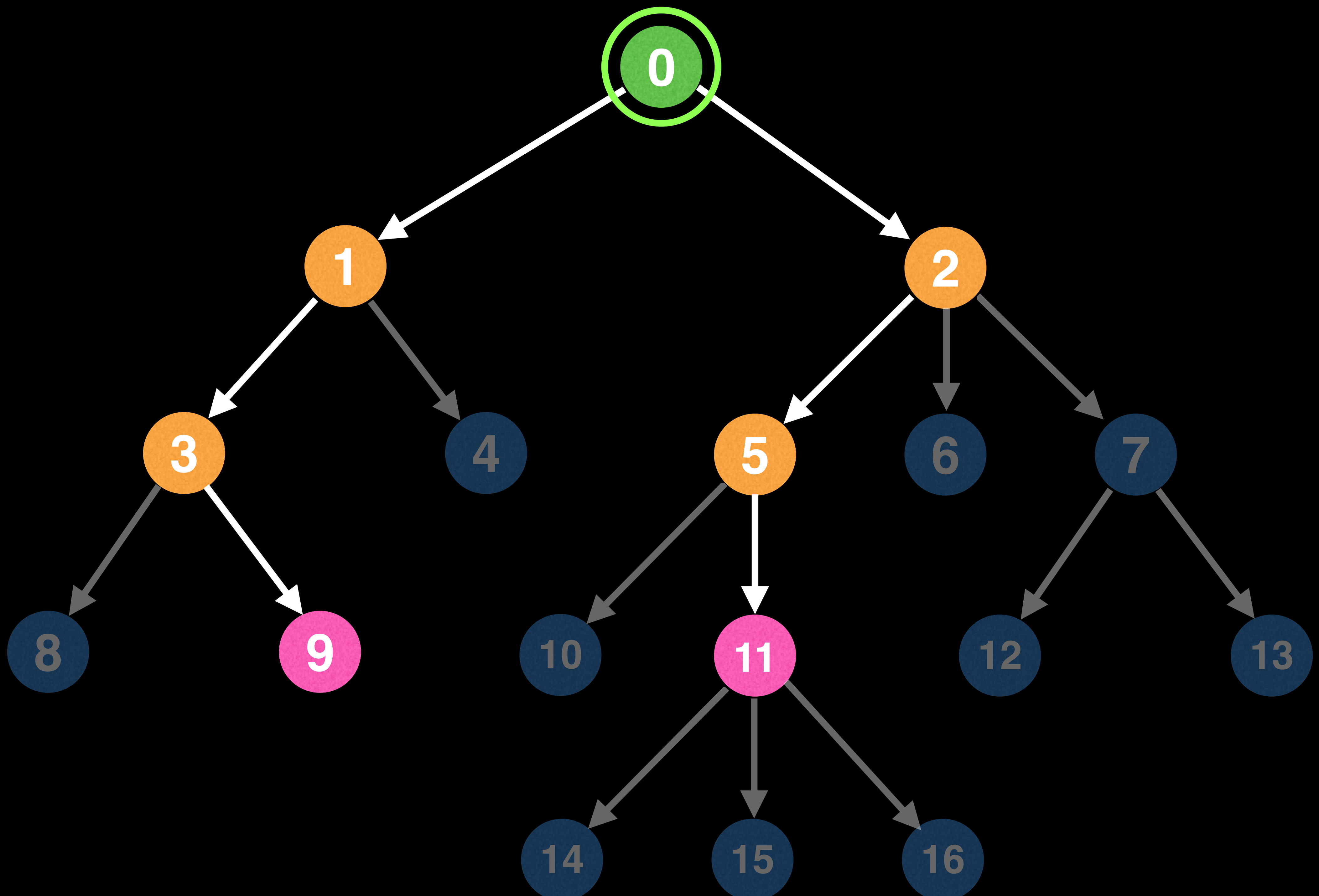
Understanding LCA

LCA(9, 11)



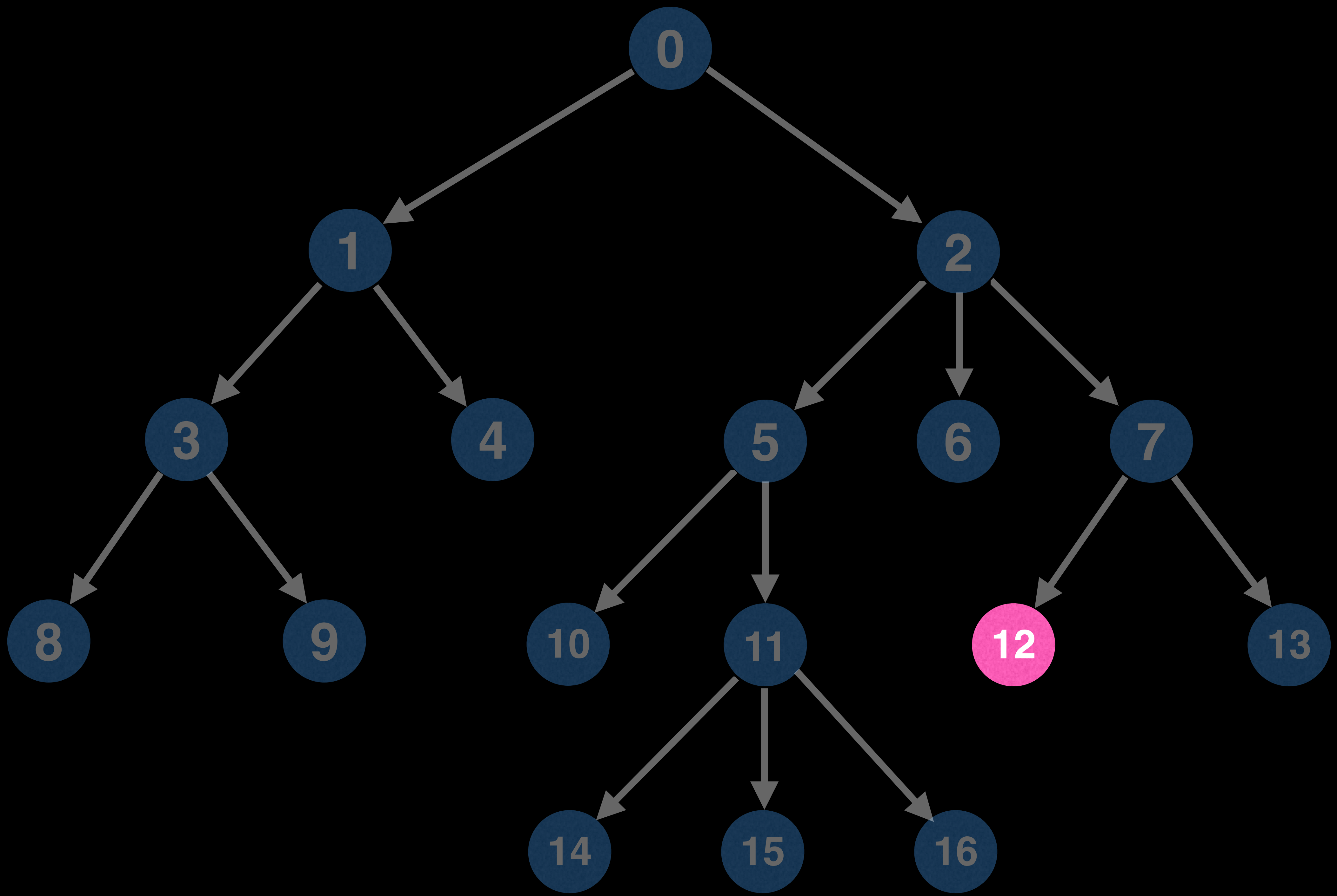
Understanding LCA

LCA(9, 11) = 0



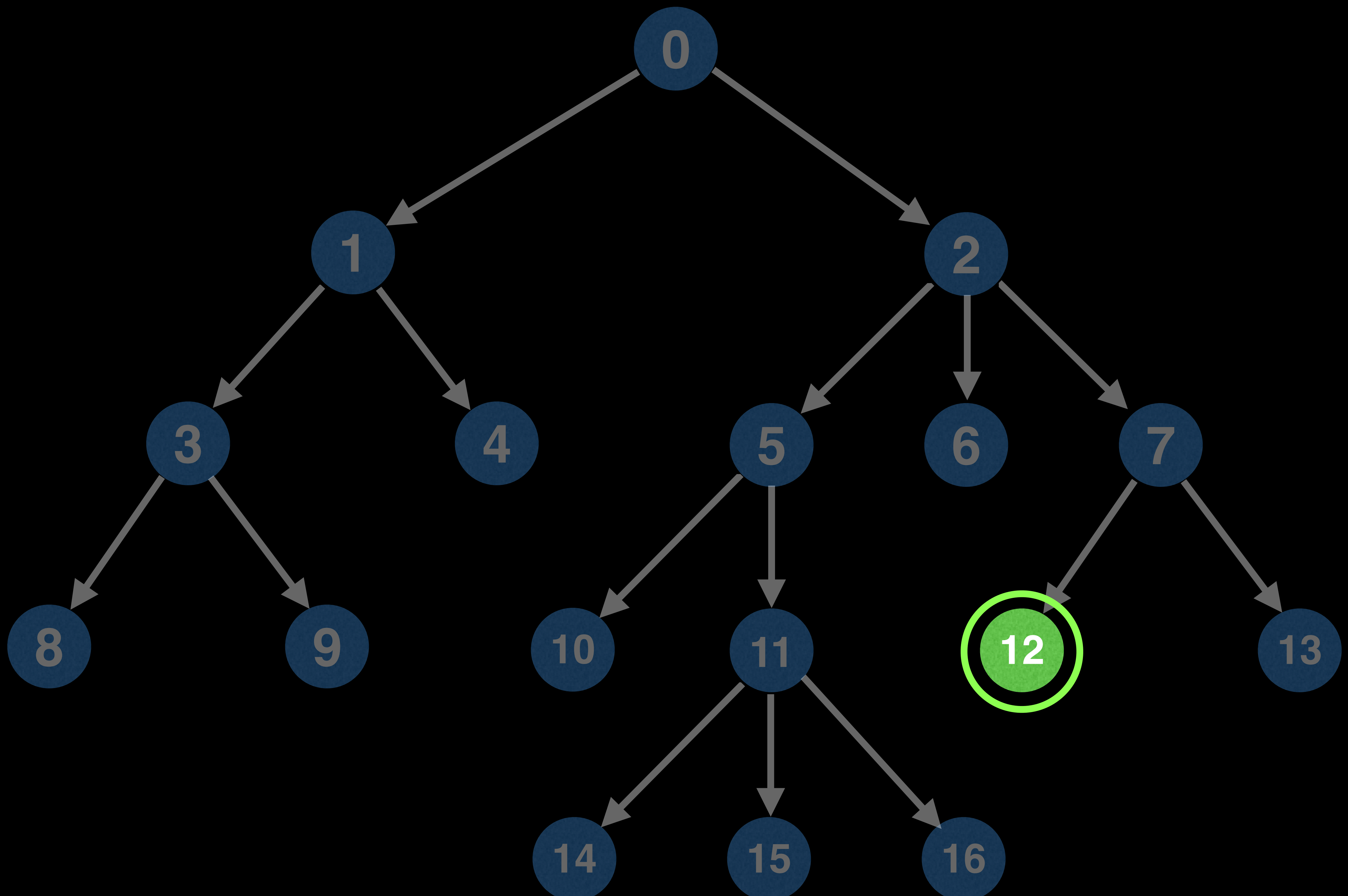
Understanding LCA

LCA(12, 12)



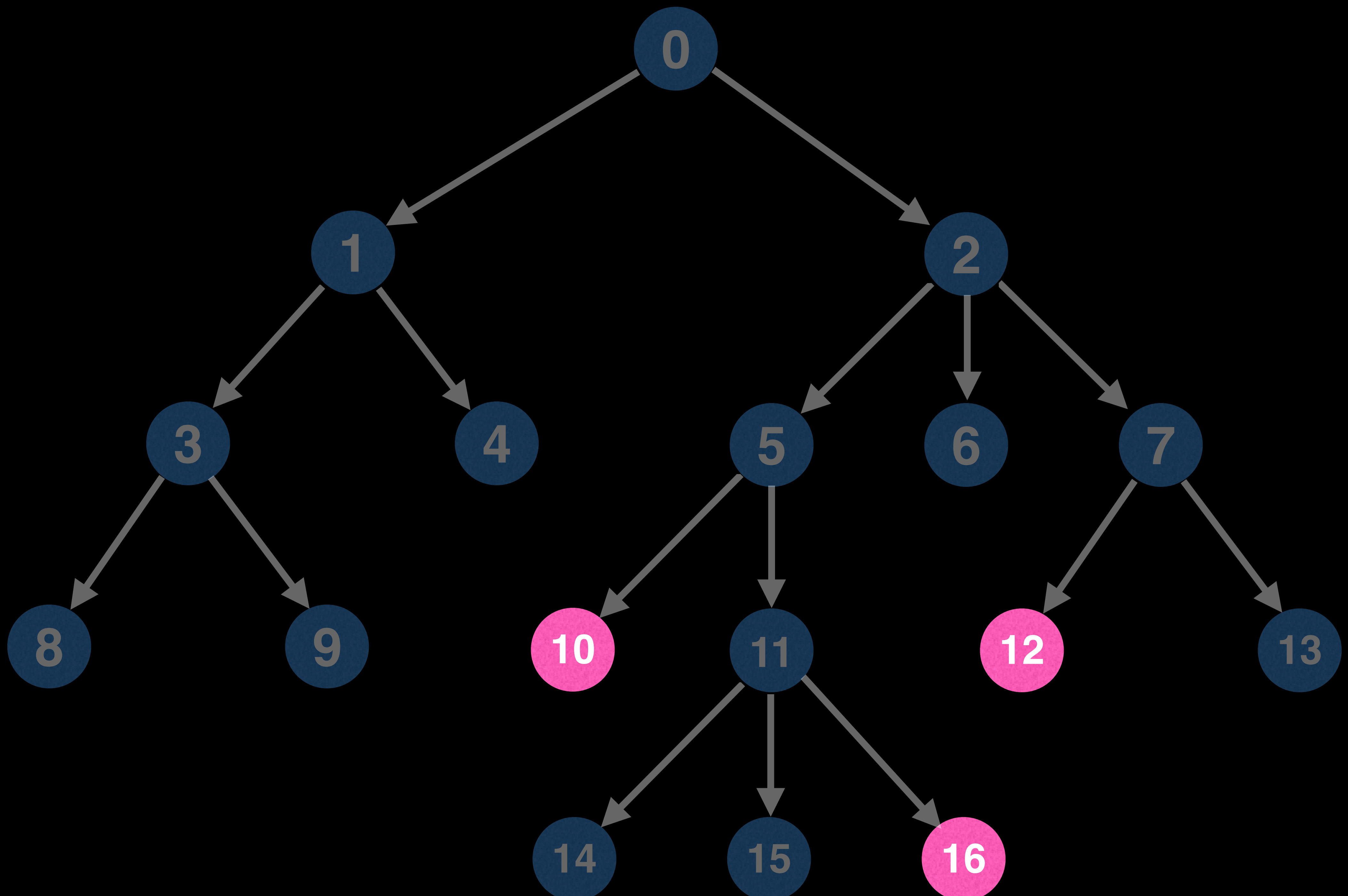
Understanding LCA

LCA(12, 12) = 12



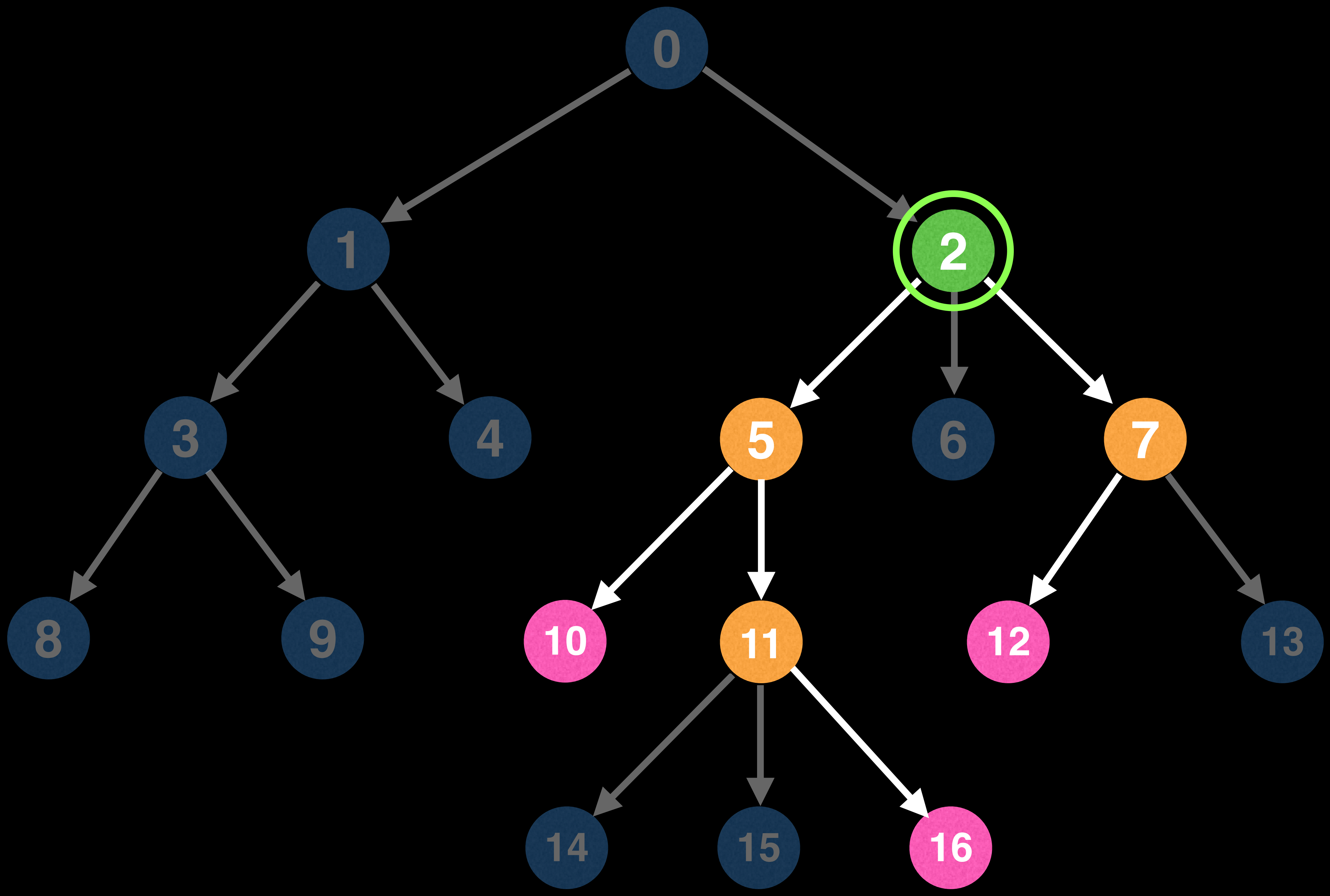
Understanding LCA

You can also find the LCA of more than 2 nodes



Understanding LCA

LCA(10, LCA(12, 16)) = 2



LCA Algorithms

There are a diverse number of popular algorithms for finding the LCA of two nodes in a tree including:

- Tarjan’s offline LCA algorithm
- Heavy-Light decomposition
- Binary Lifting
- etc…

Today, we’re going to cover how to find the LCA using the **Eulerian tour** + **Range Minimum Query (RMQ)** method.

LCA Algorithms

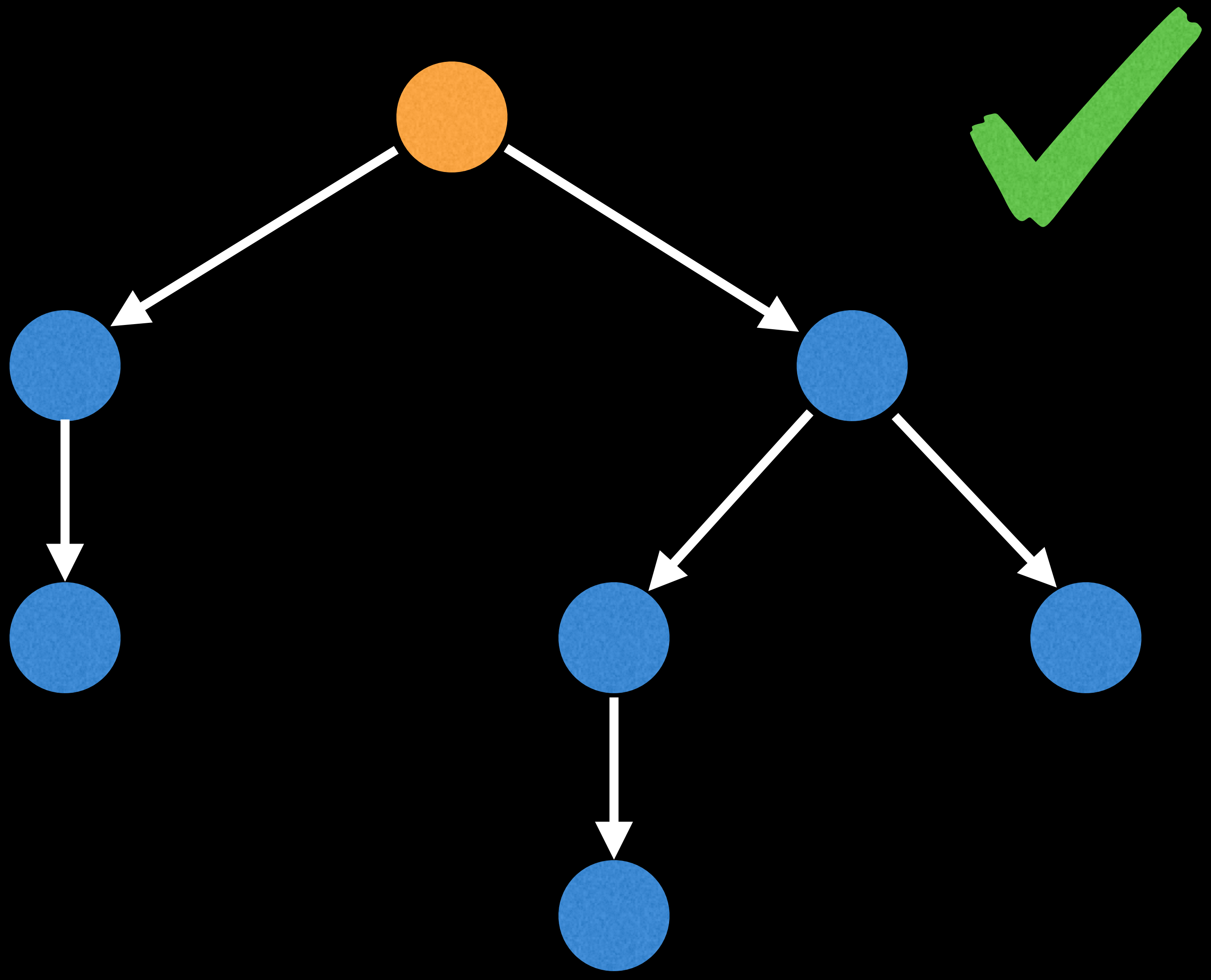
There are a diverse number of popular algorithms for finding the LCA of two nodes in a tree including:

- Tarjan’s offline LCA algorithm
- Heavy-Light decomposition
- Binary Lifting
- etc…

Today, we’re going to cover how to find the LCA using the **Eulerian tour** + **Range Minimum Query (RMQ)** method.

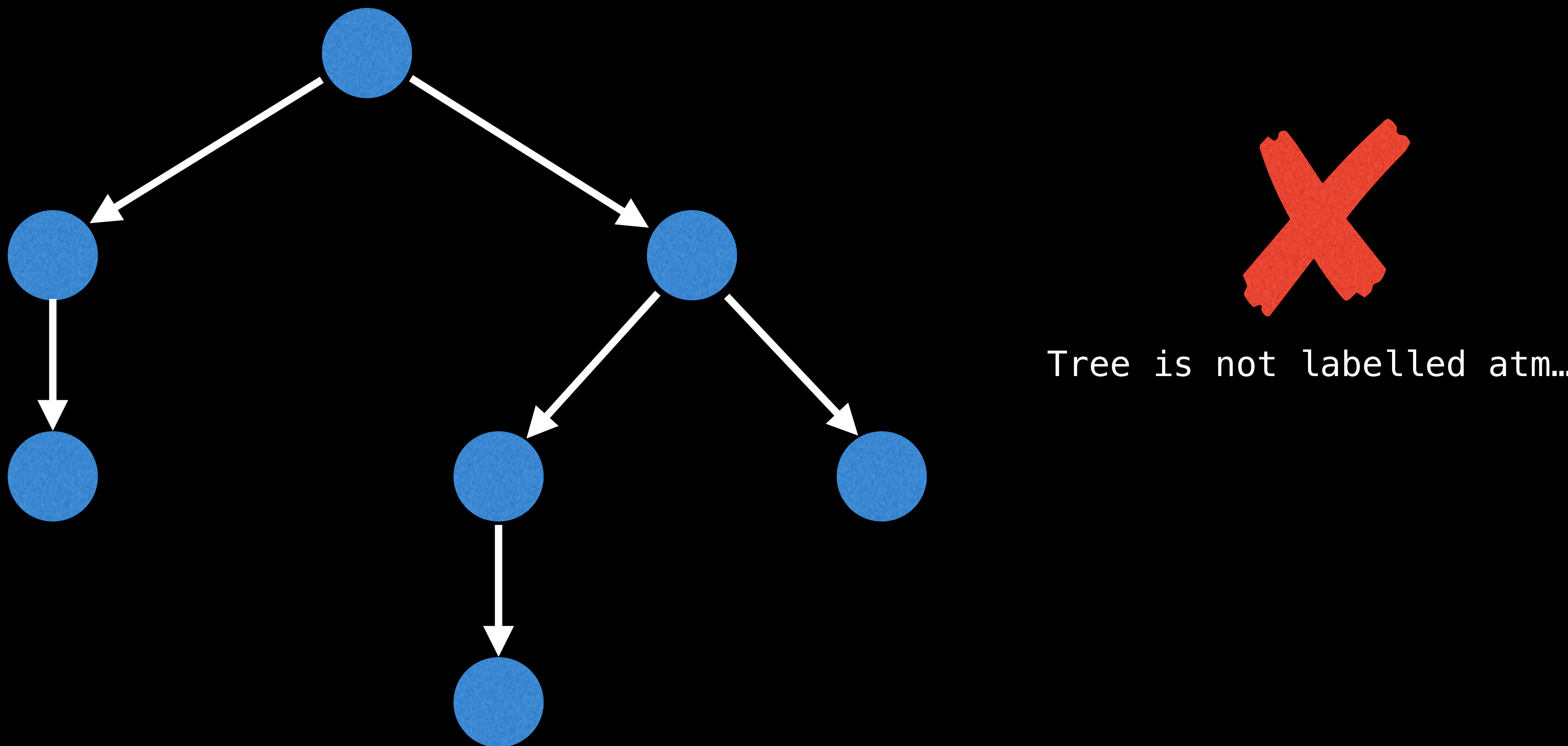
This method can answer LCA queries in **O(1)** time with **O(nlogn)** pre-processing when using a **Sparse Table** to do the RMQs.

However, the pre-processing time can be improved to **O(n)** with the Farach-Colton and Bender optimization.



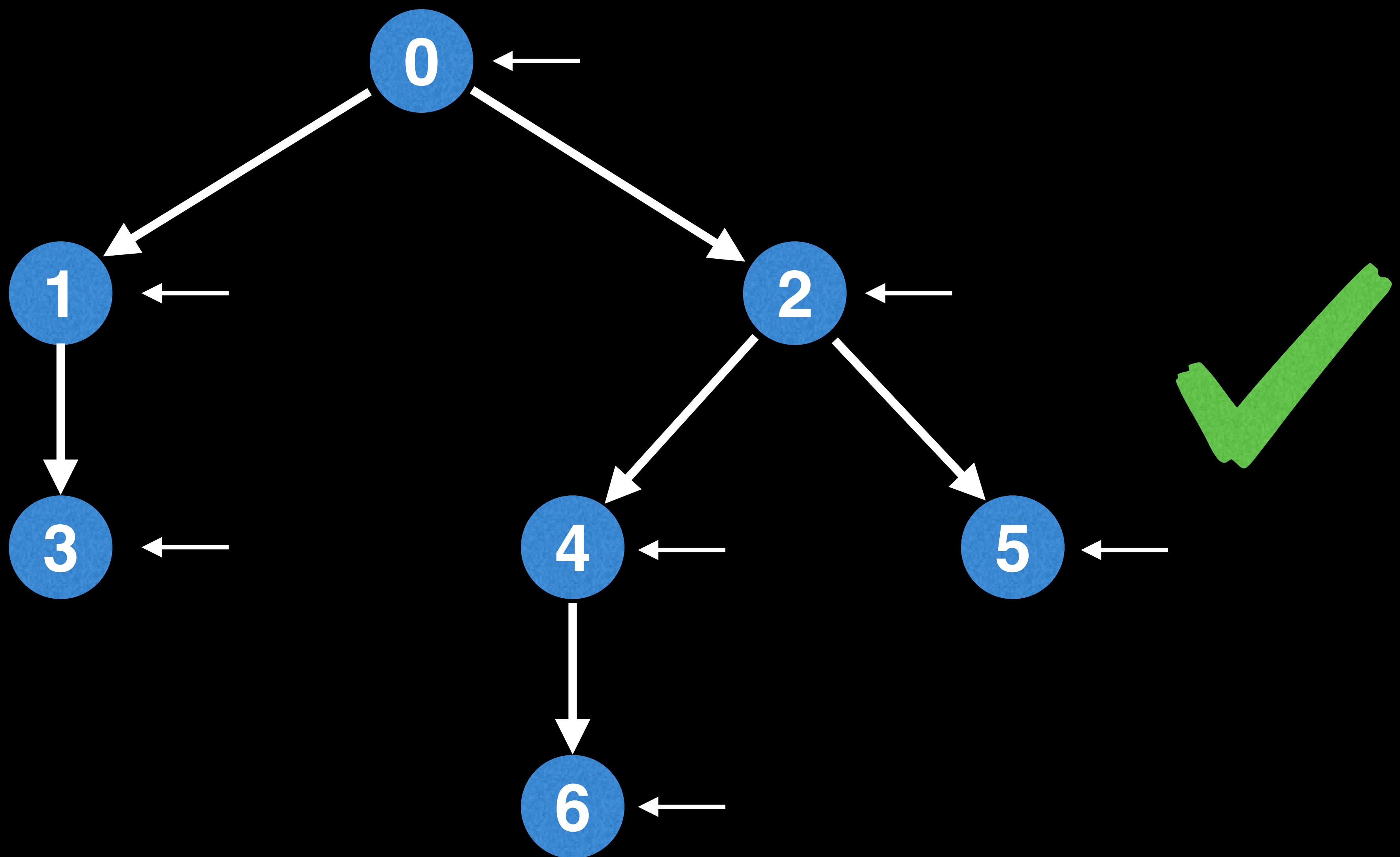
Given a tree we want to do LCA queries on, first:

1. Make sure the tree is **rooted**



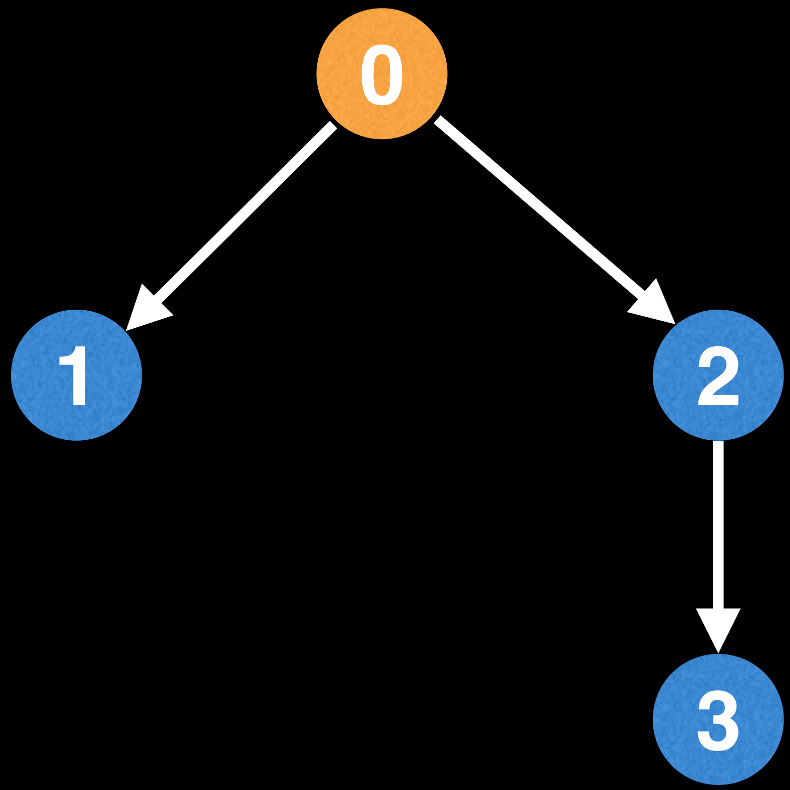
Given a tree we want to do LCA queries on, first:

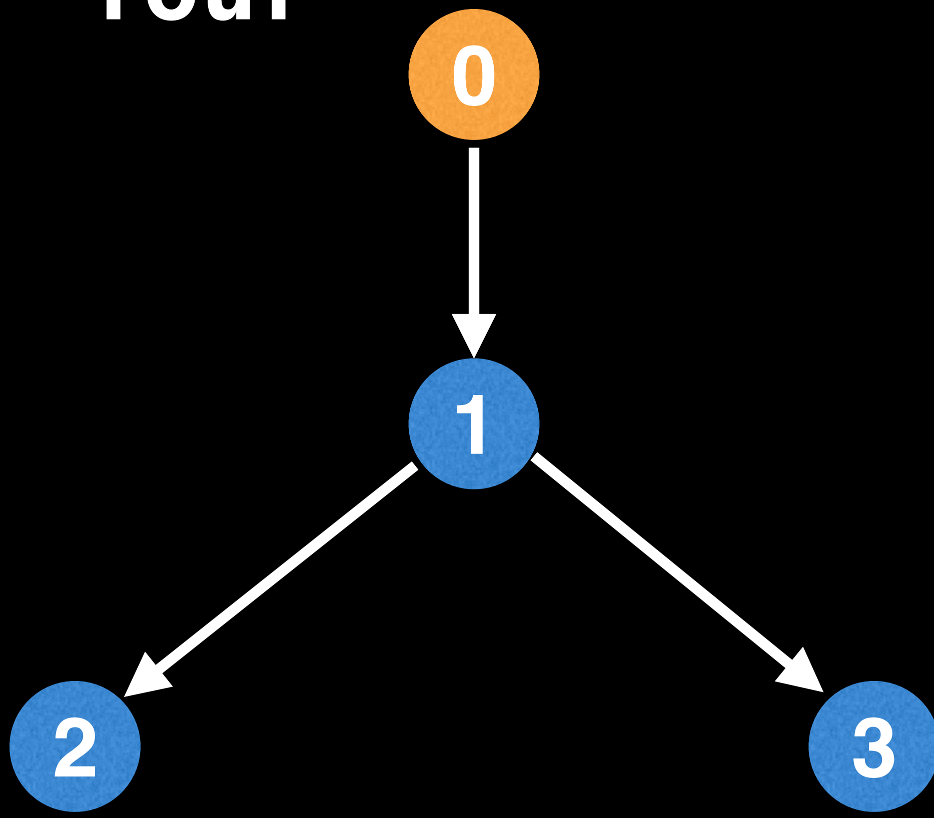
2. Ensure that all nodes are **uniquely indexed** in some way so that we can reference them later.



One easy way to index each node is by assigning each node a unique id between [0, n-1]

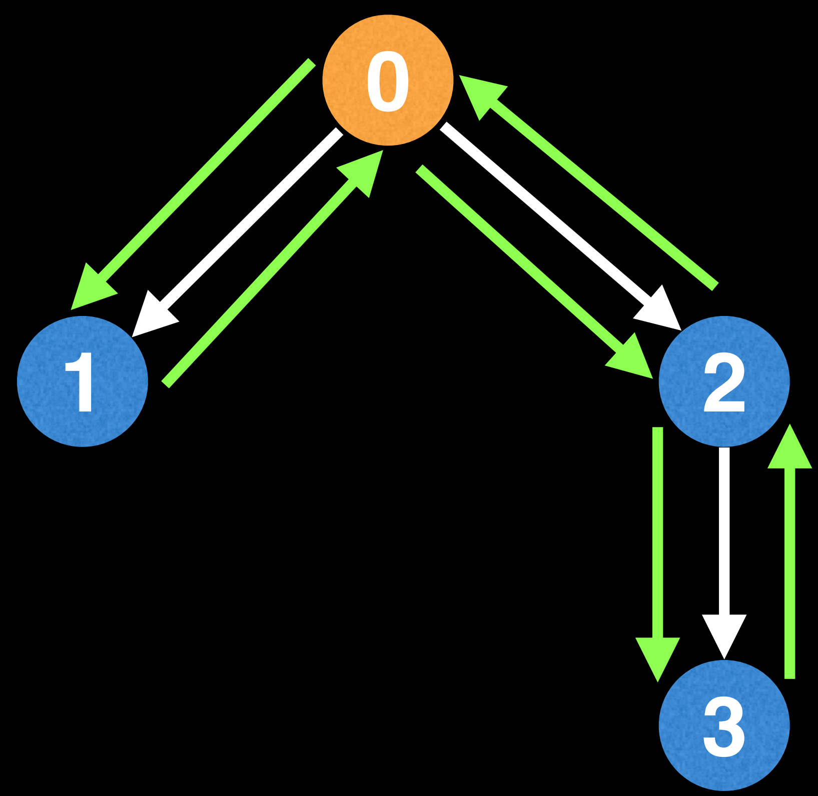
LCA with Euler Tour

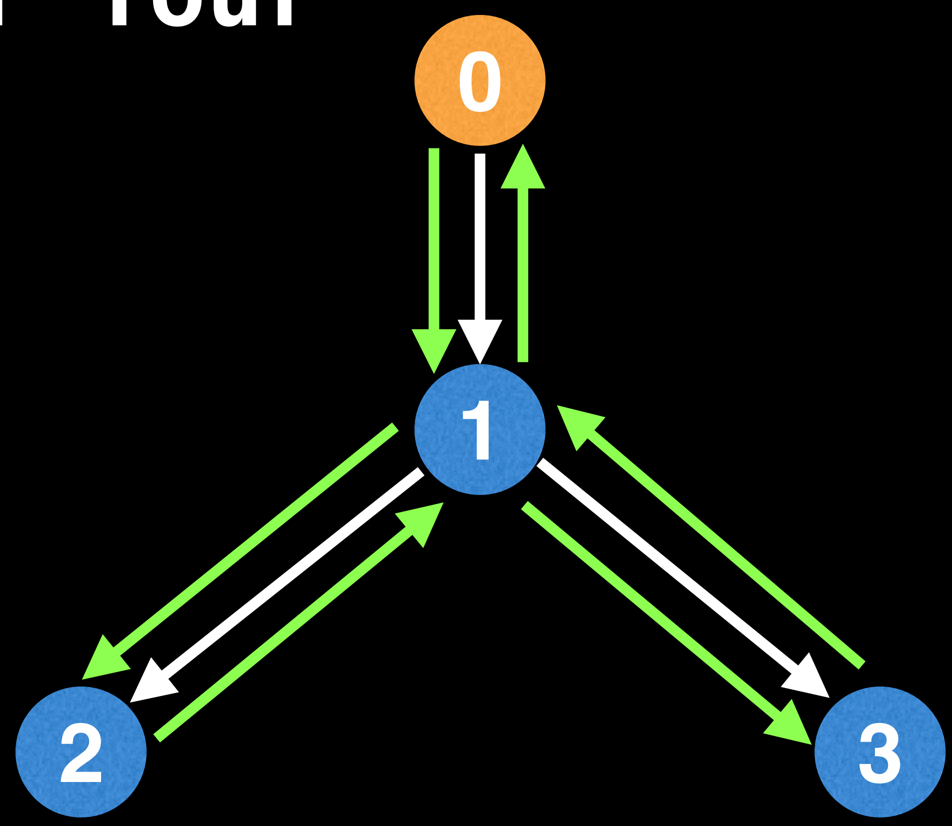




As you might have guessed, the Eulerian tour method begins by finding an Eulerian tour of the edges in a rooted tree.

LCA with Euler Tour

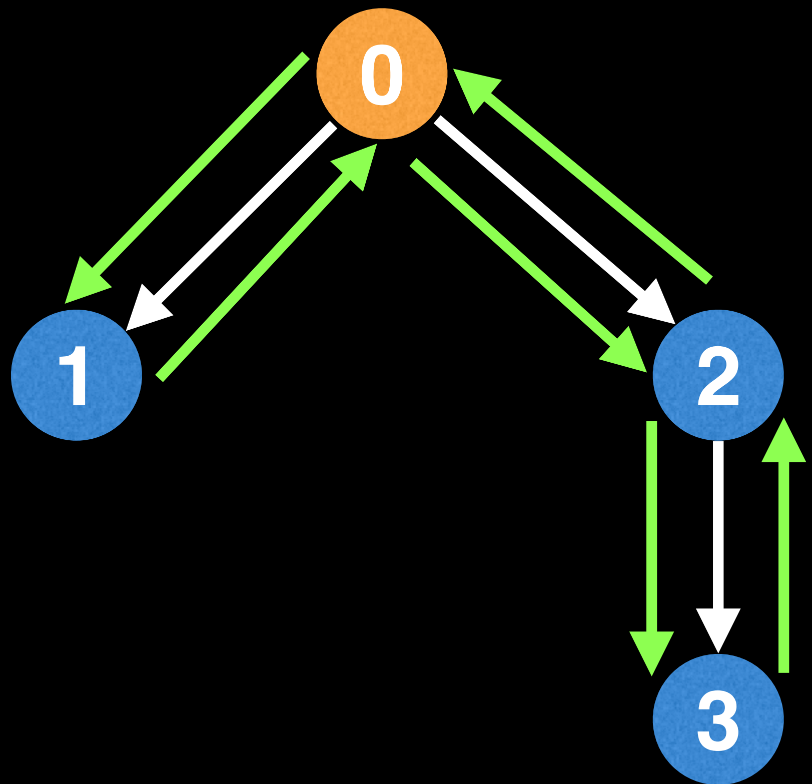




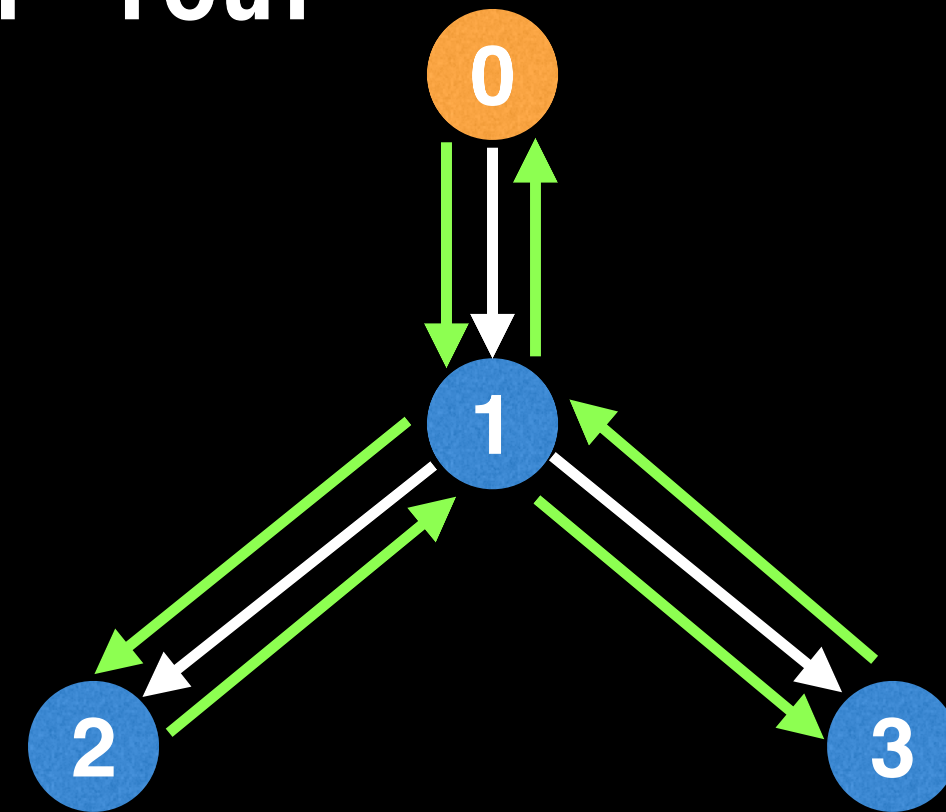
As you might have guessed, the Eulerian tour method begins by finding an Eulerian tour of the edges in a rooted tree.

Rather than doing the Euler tour on the white edges of our tree, we're going to do the Euler tour on a new set of imaginary **green edges** which wrap around the tree. This ensures that our tour visits every node in the tree.

LCA with Euler Tour

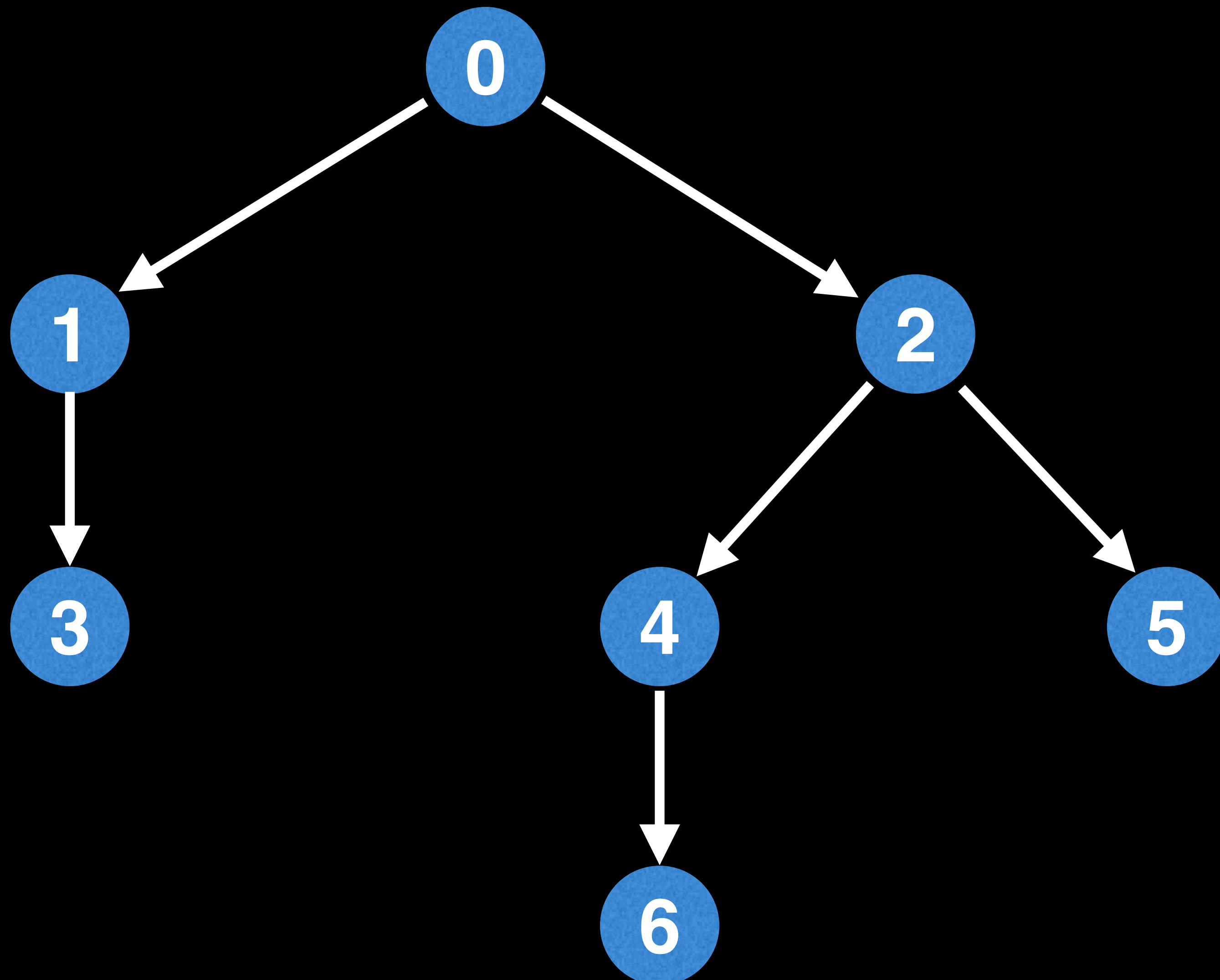


tour: [0,1,0,2,3,2,0]



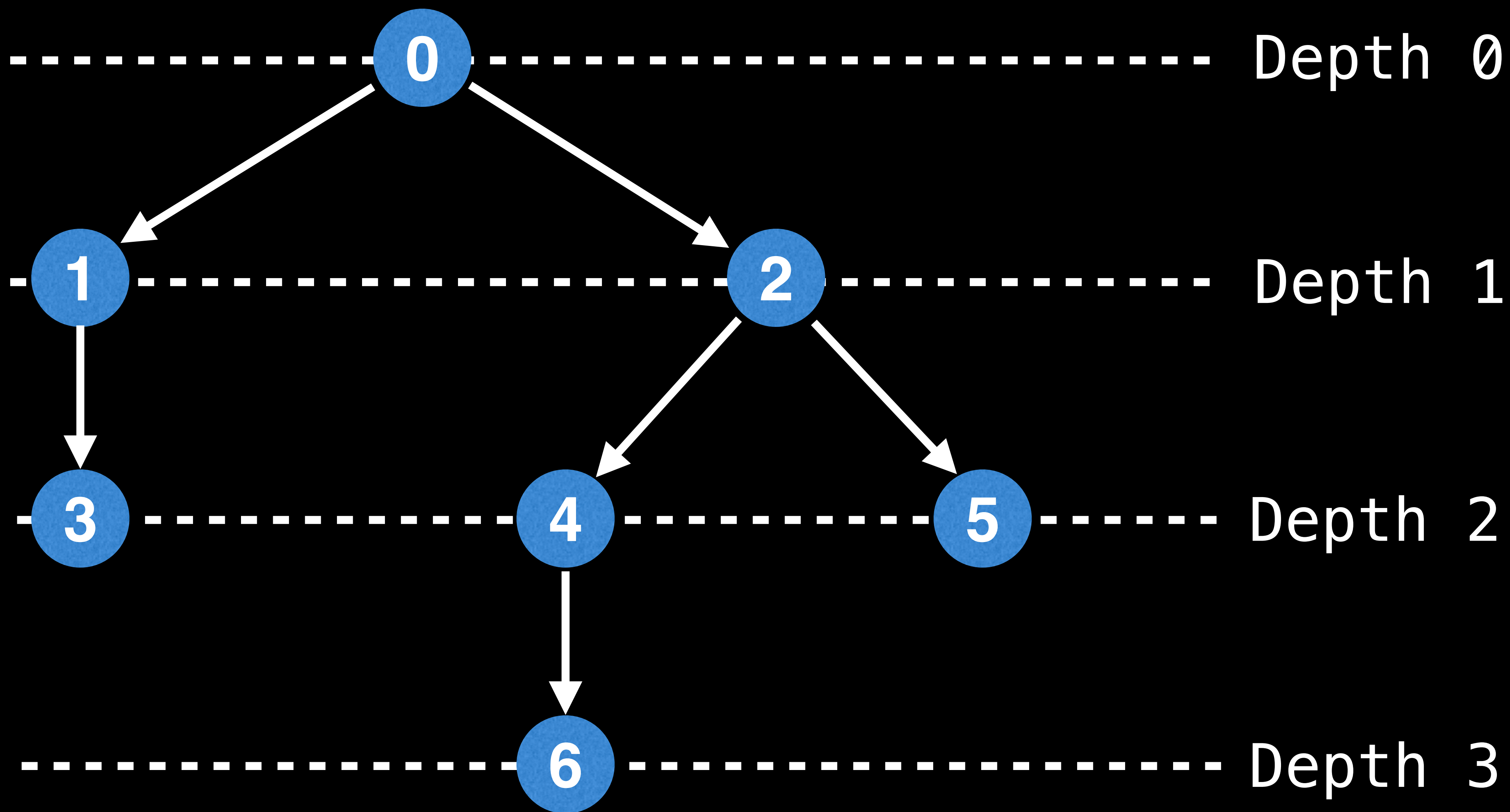
tour: [0,1,2,1,3,1,0]

Start an **Eulerian tour** (Eulerian circuit) at the root node, traverse all green edges, and finally return to the root node. As you do this, keep track of which nodes you visit and this will be your Euler tour.



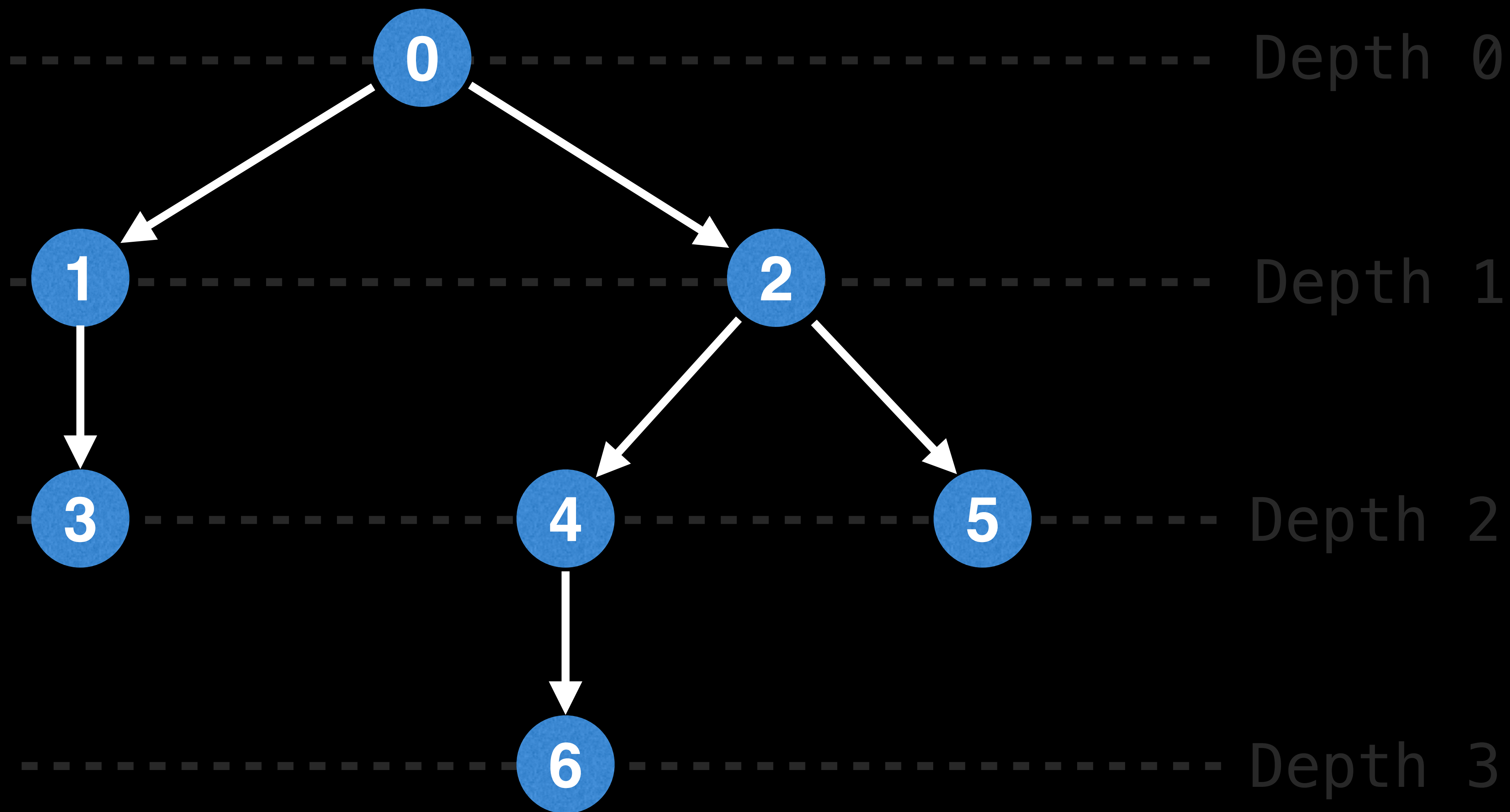
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth													

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes													



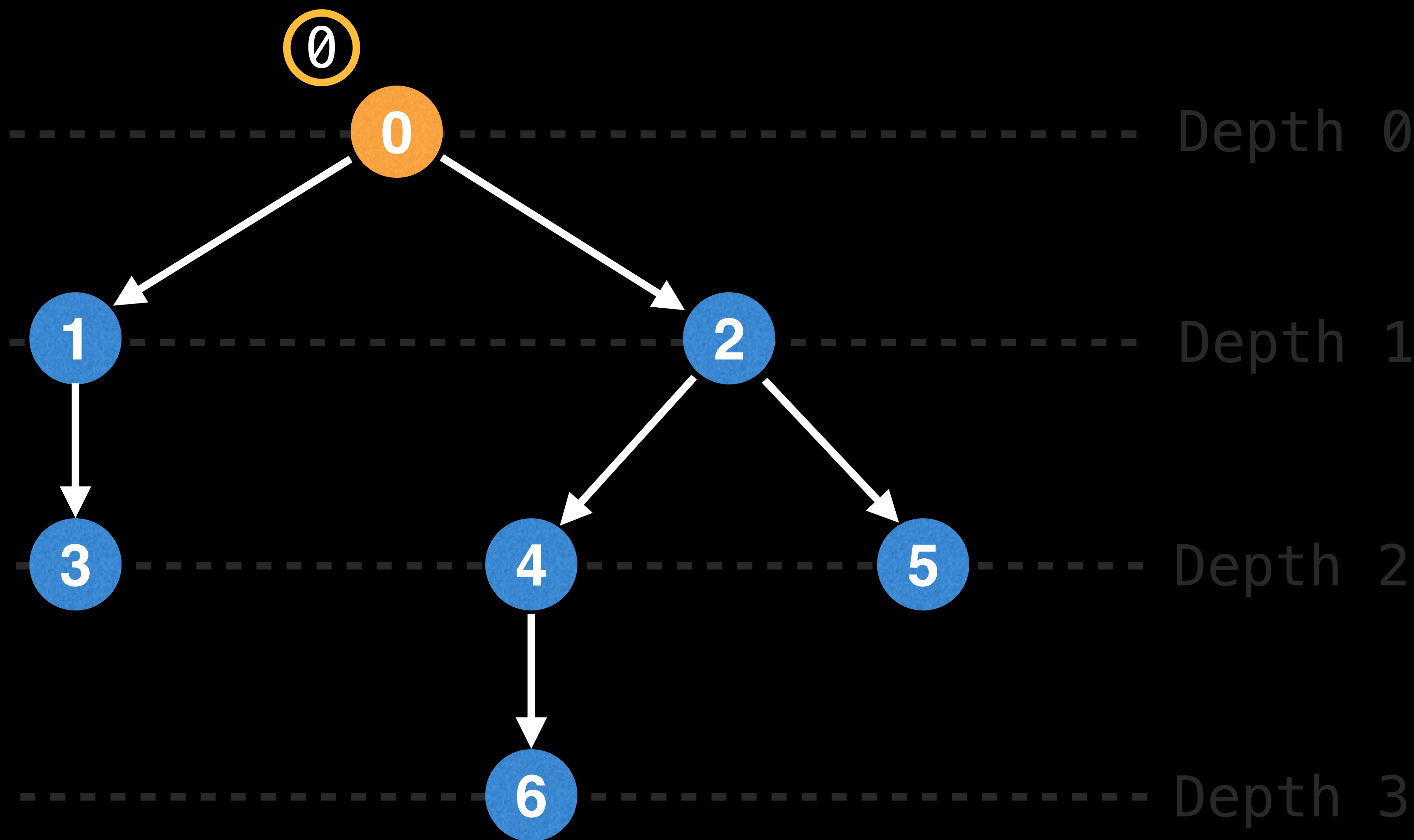
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth													

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes													



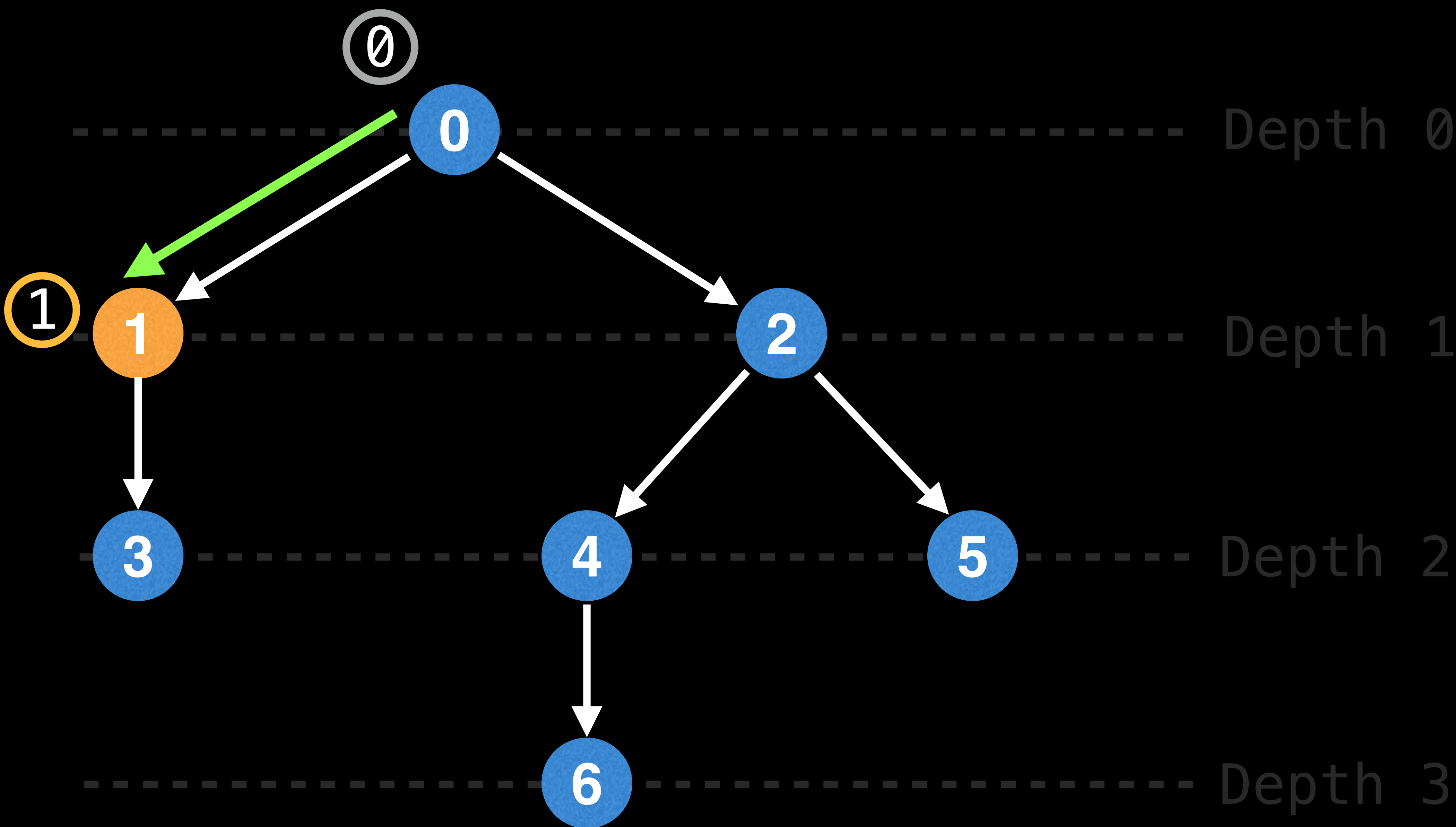
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth													

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes													



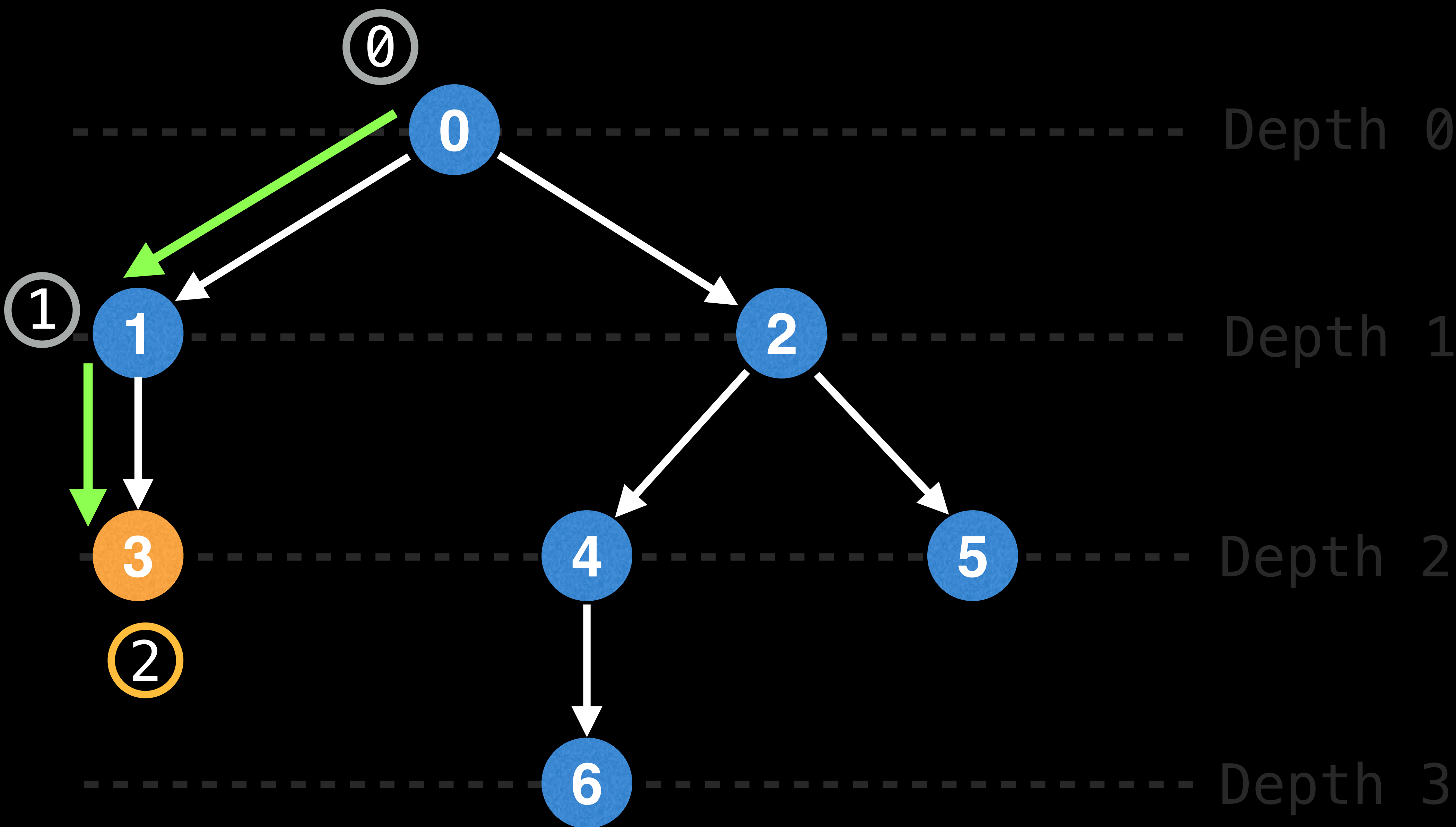
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0												

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0												



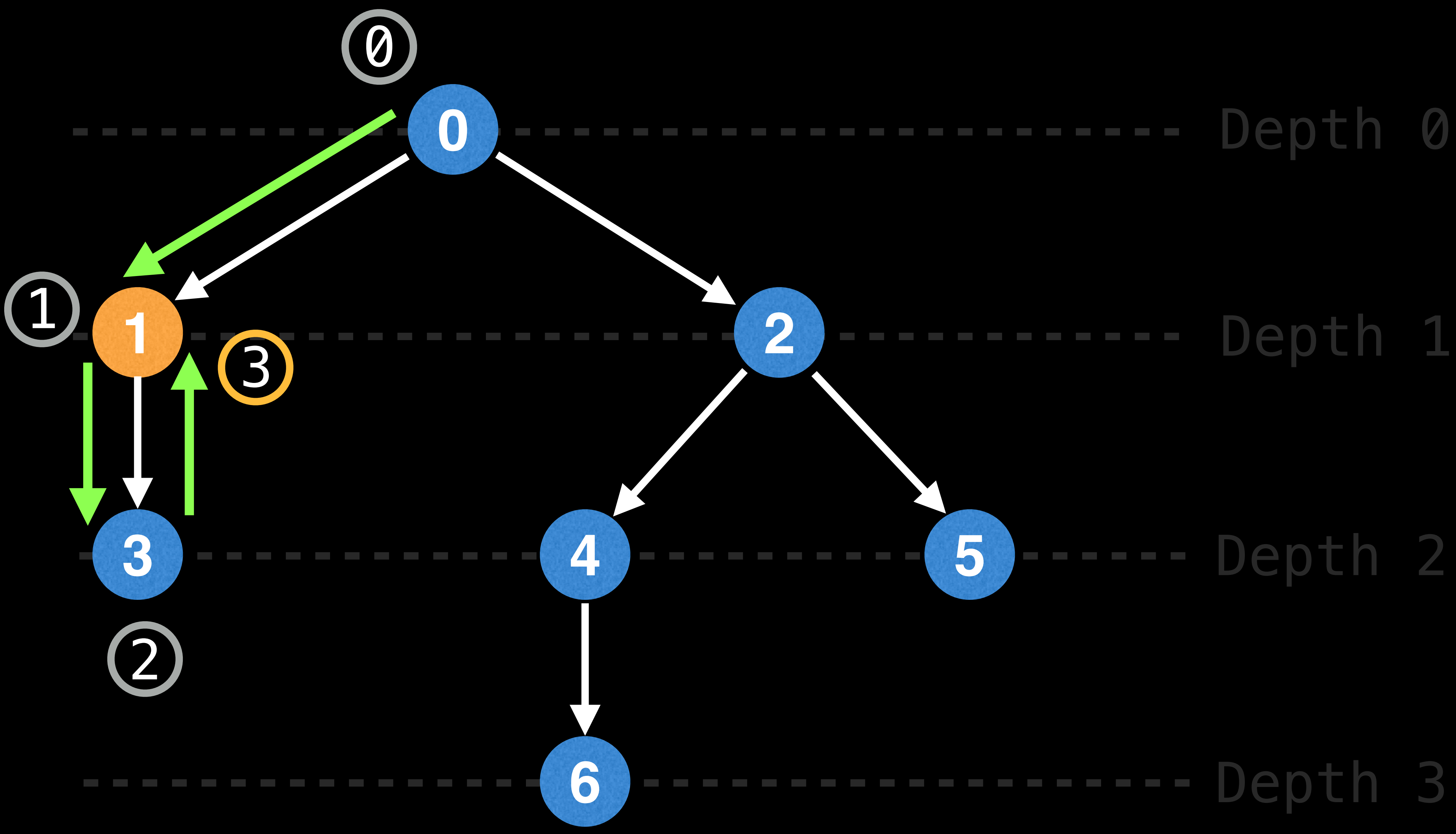
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1											

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1											



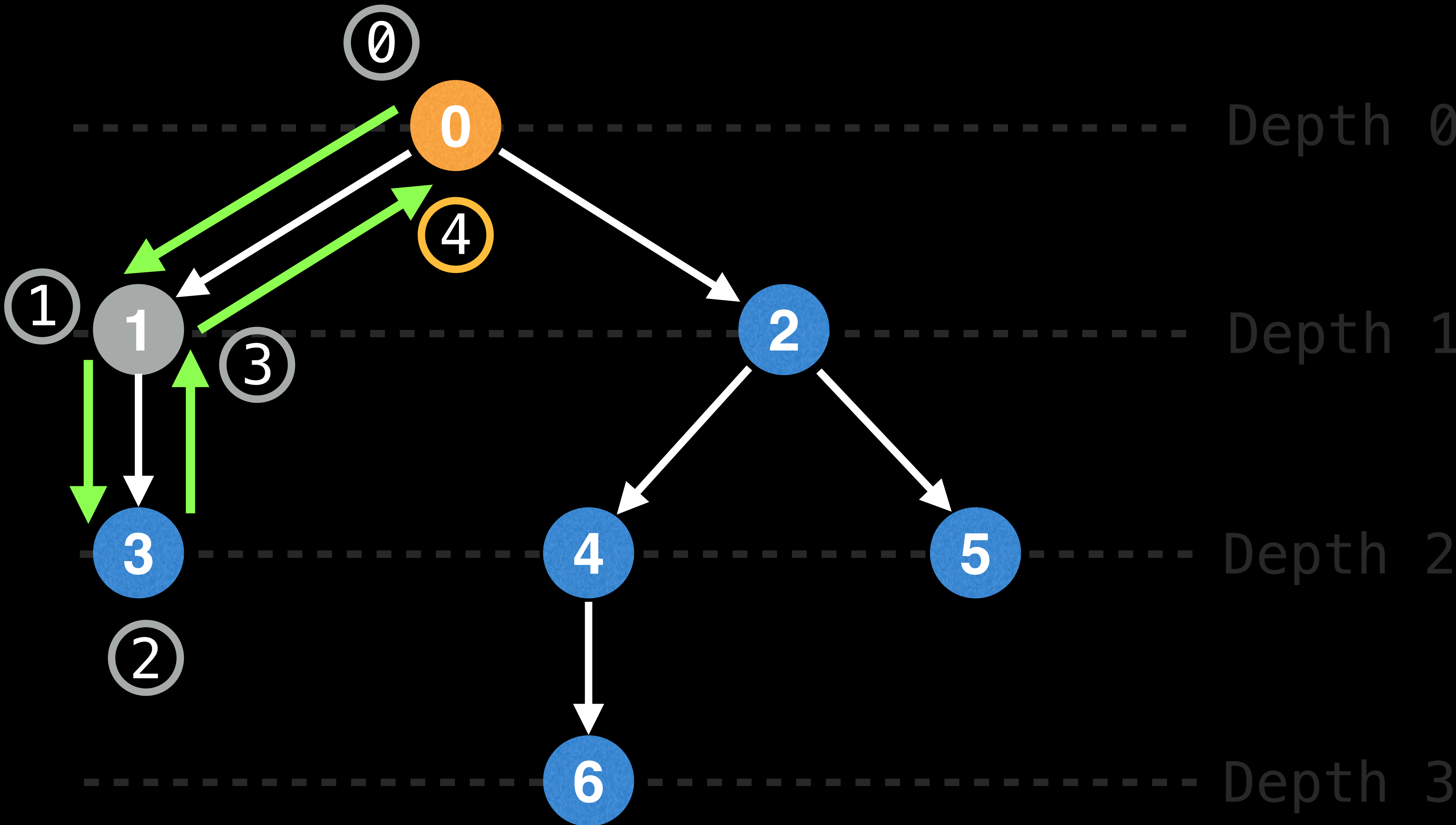
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2										

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3										



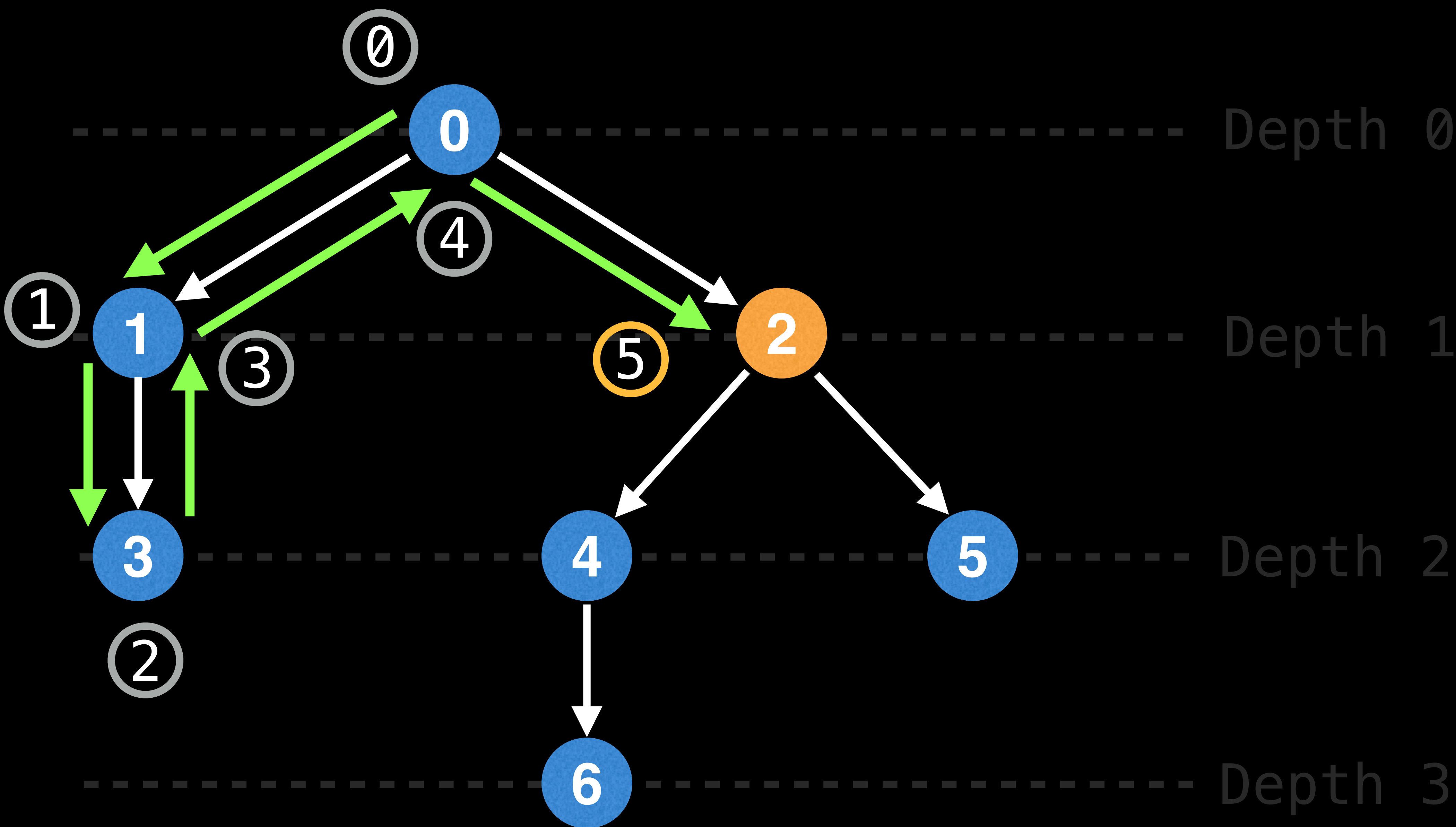
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1									

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1									



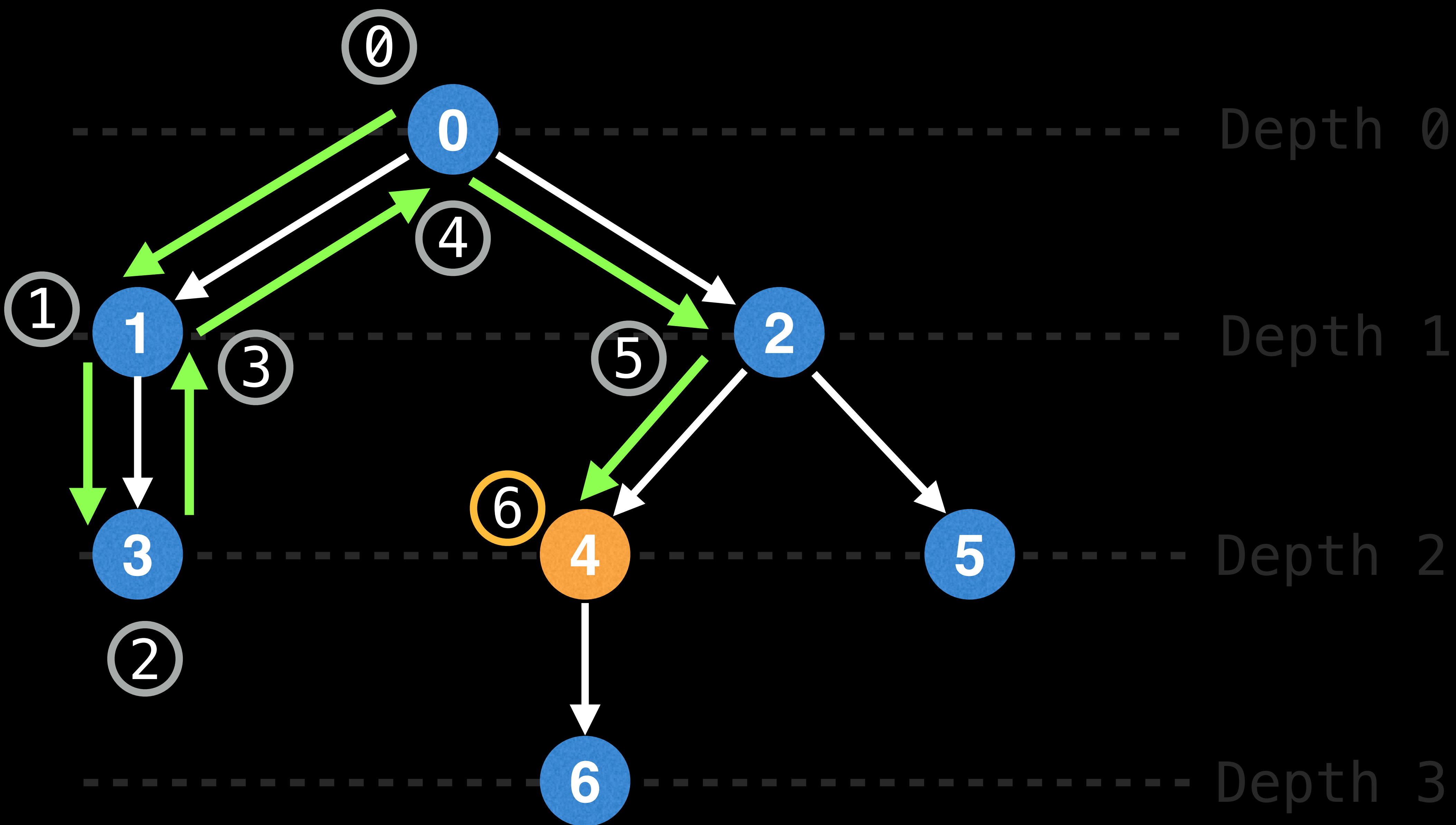
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0								

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0								



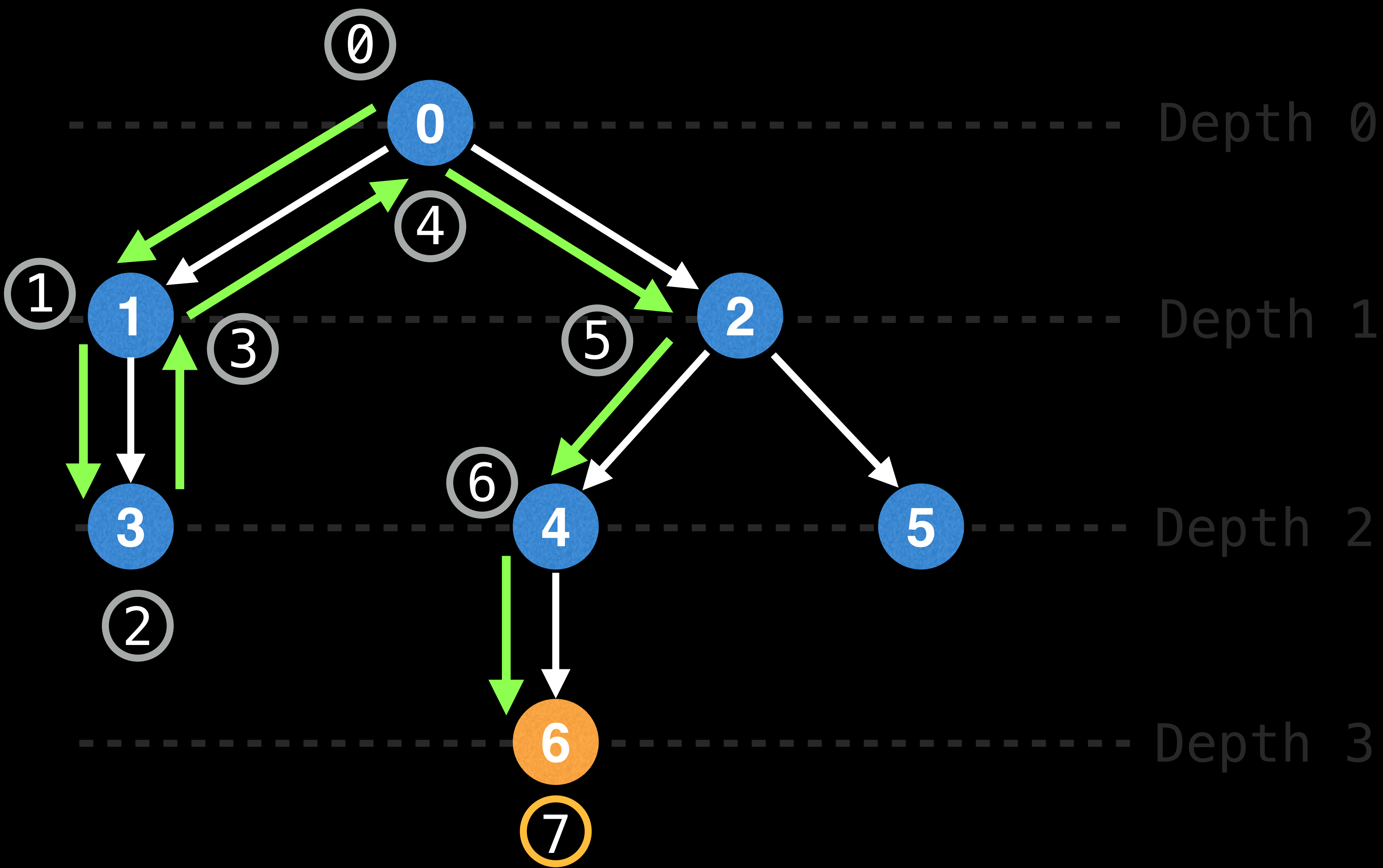
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1							

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2							



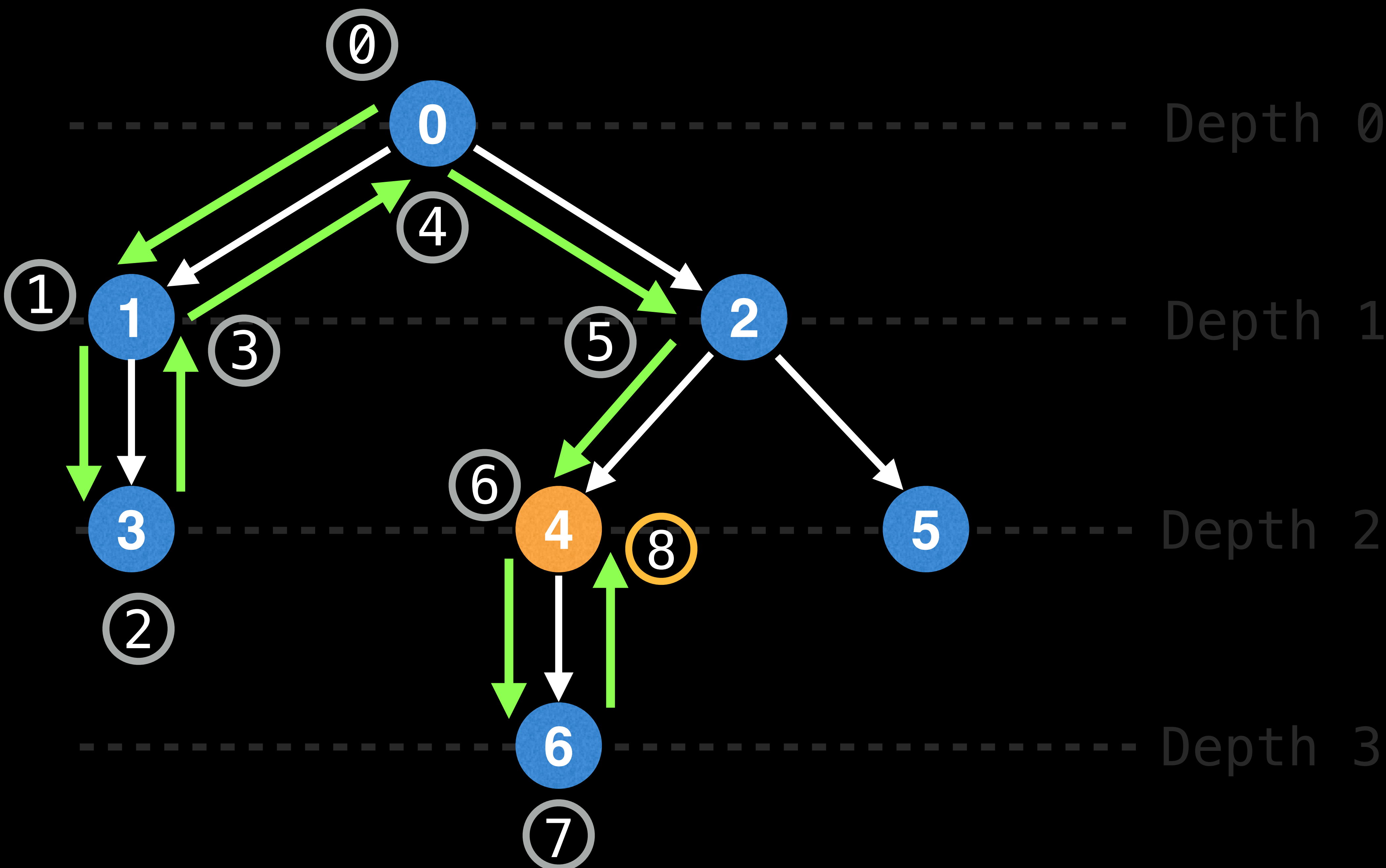
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2						

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4						

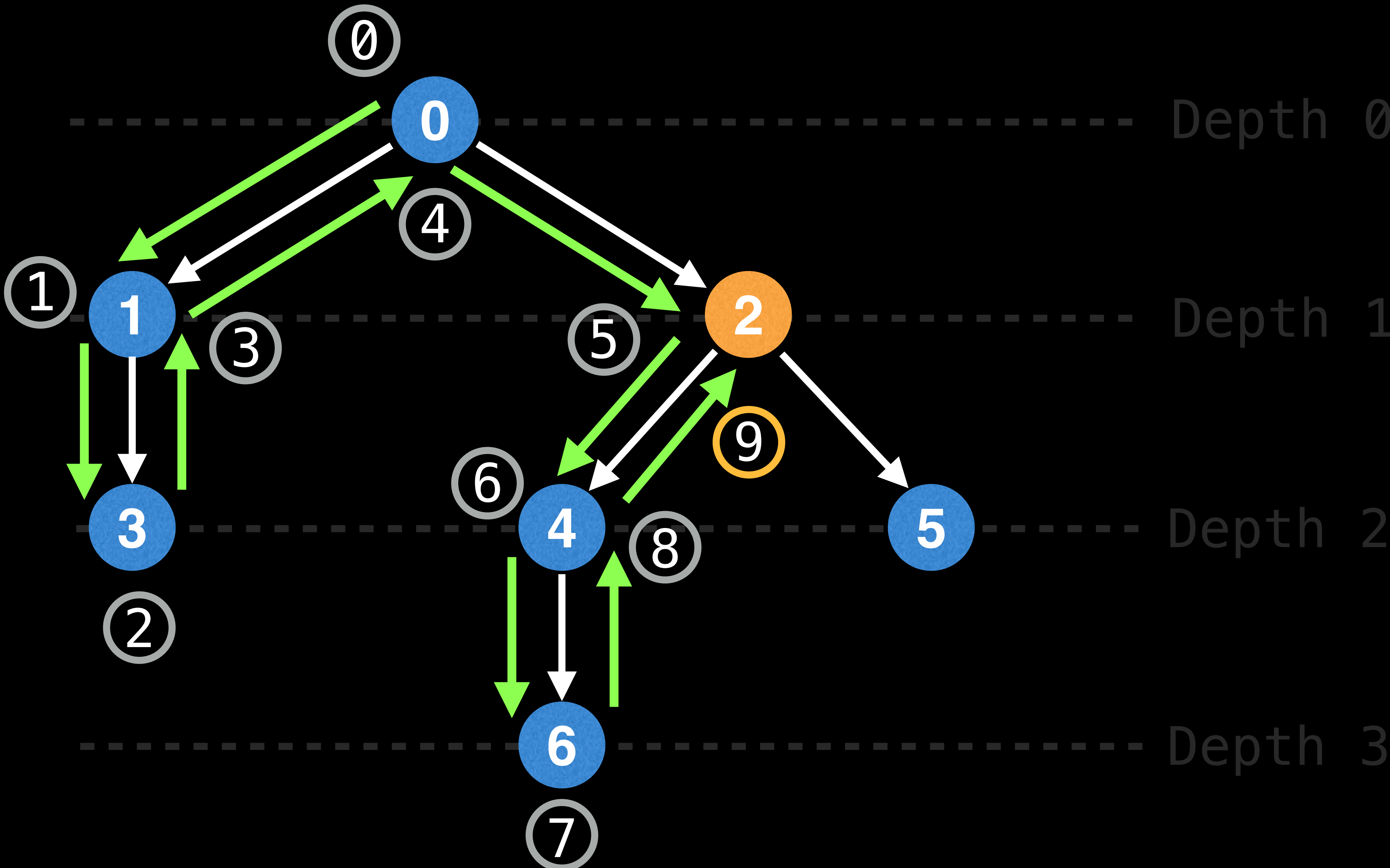


	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3					

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6					

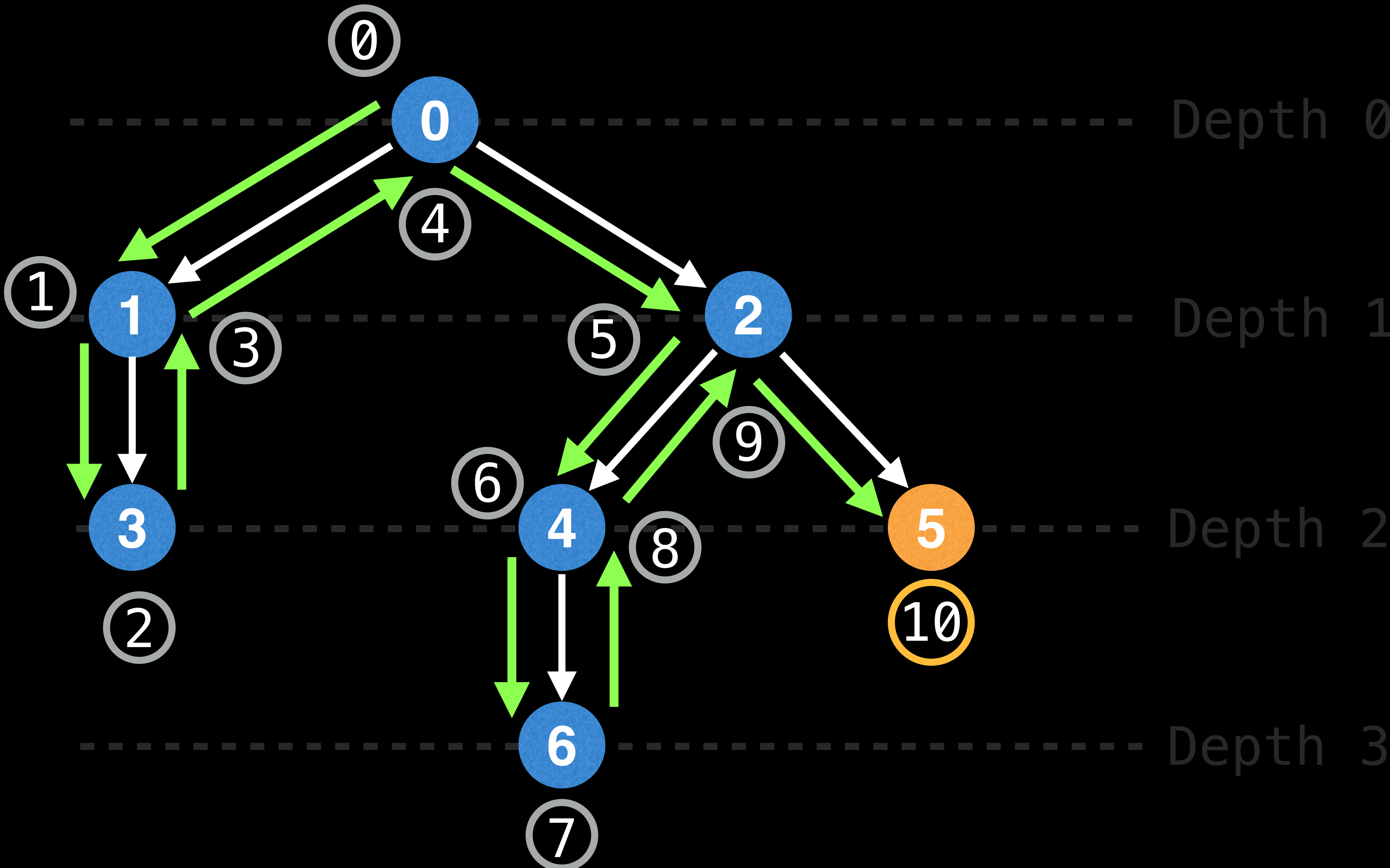


	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2				
nodes	0	1	3	1	0	2	4	6	4				



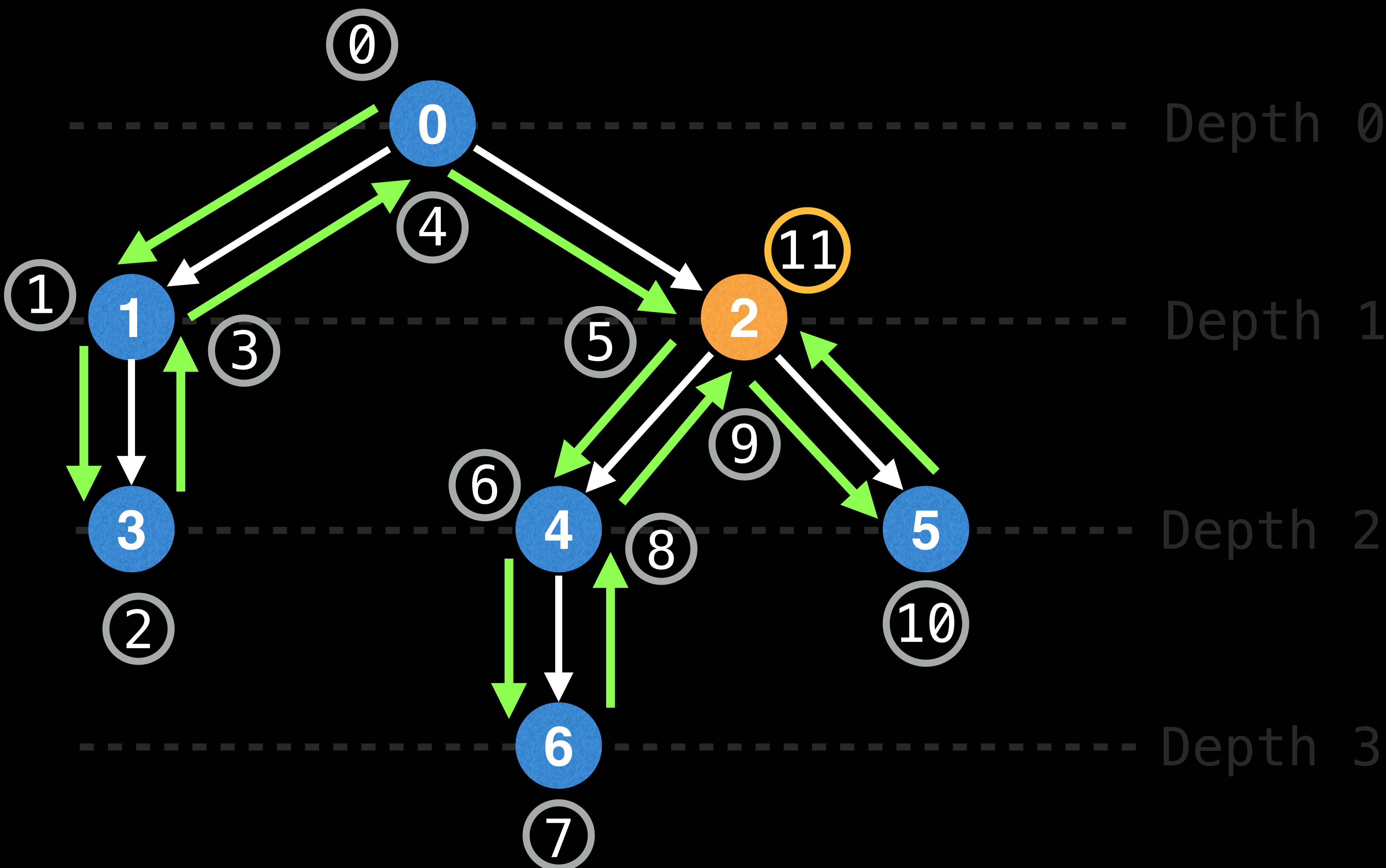
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1			

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2			



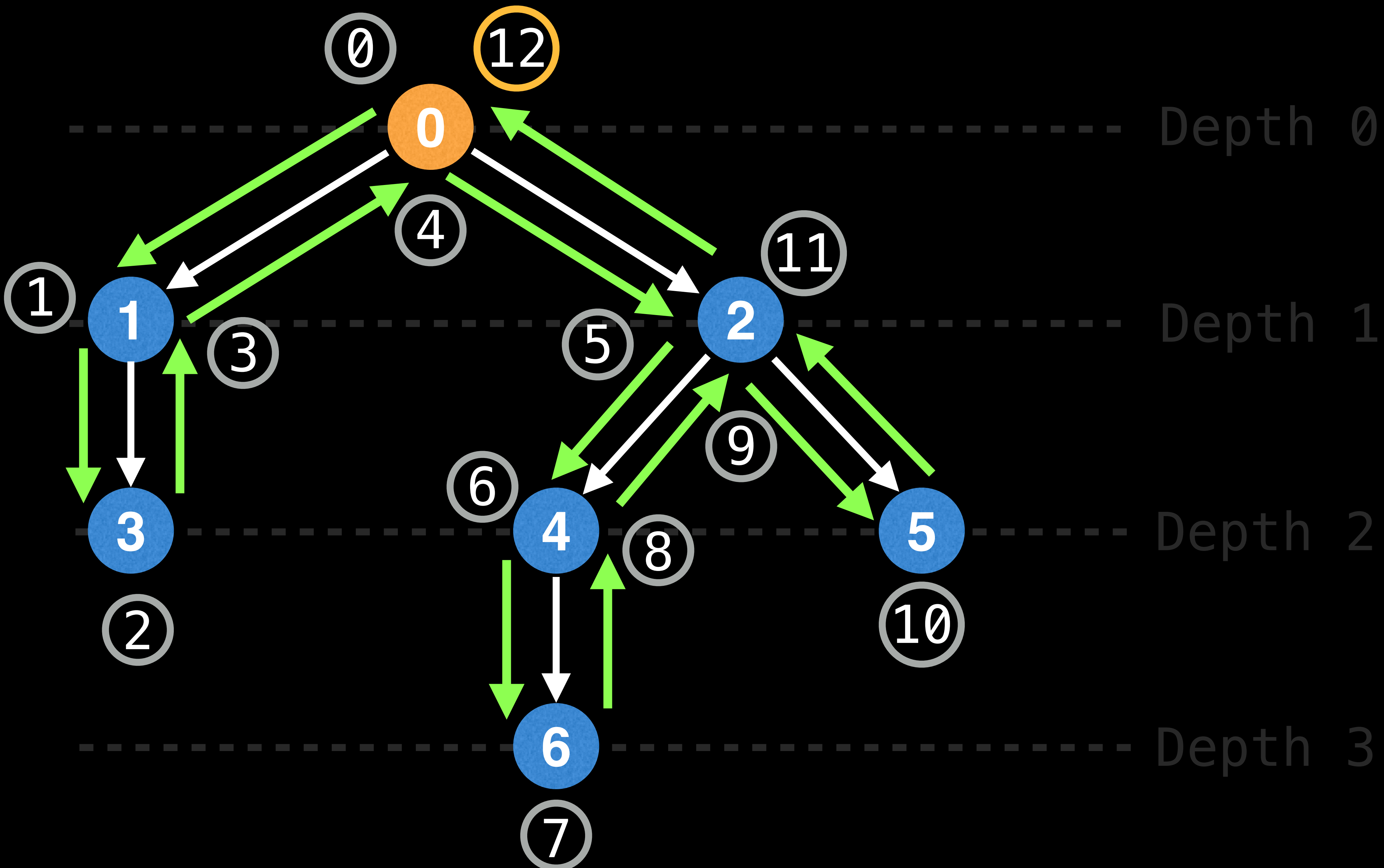
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2		

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5		



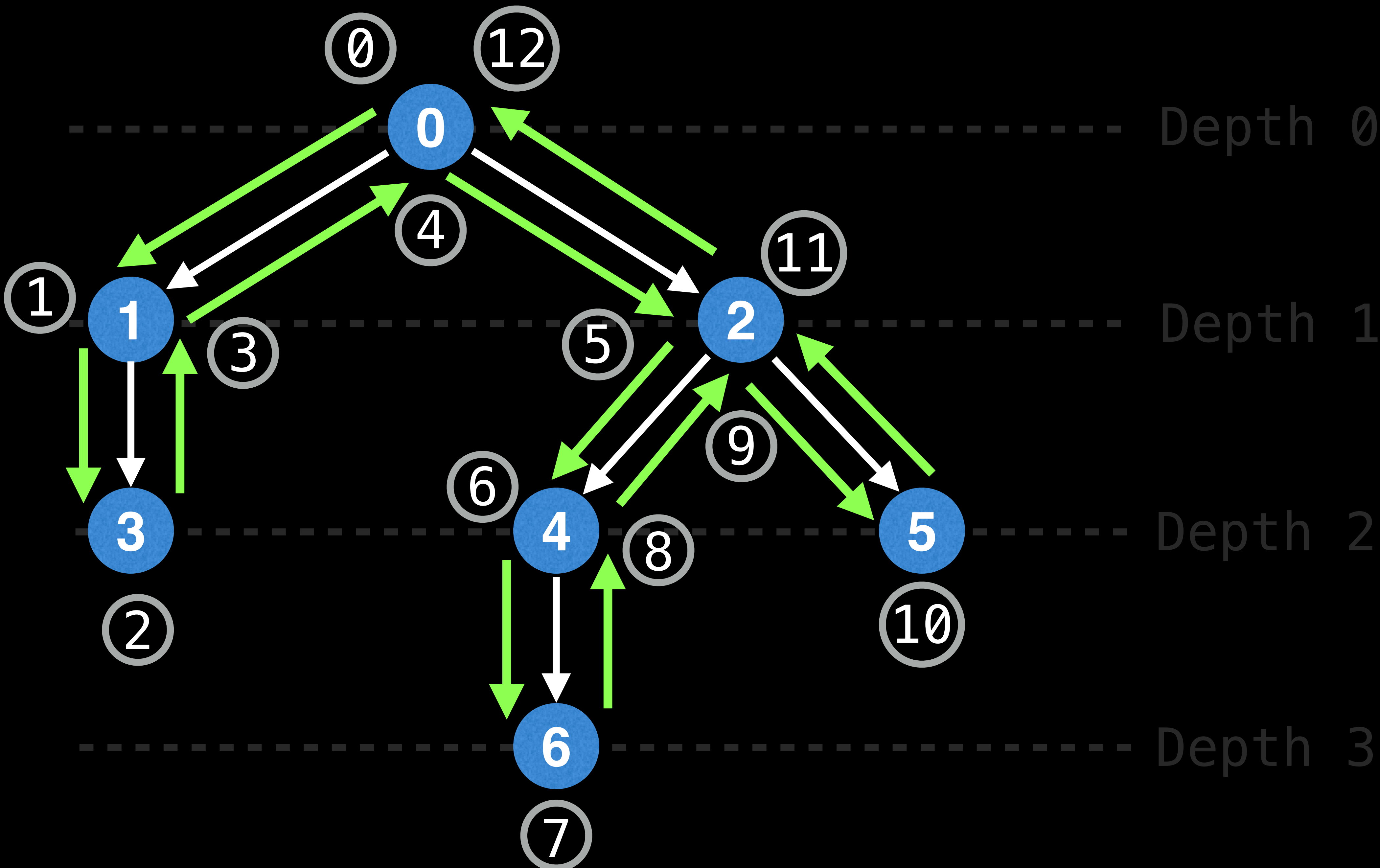
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5	2	



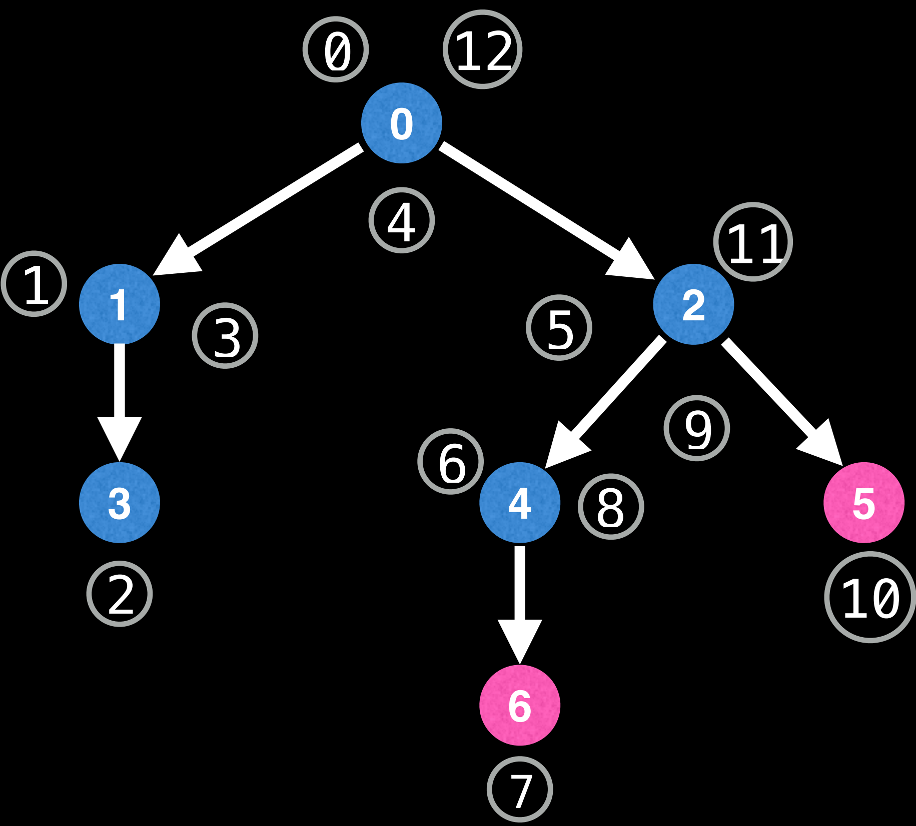
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5	2	0



	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0

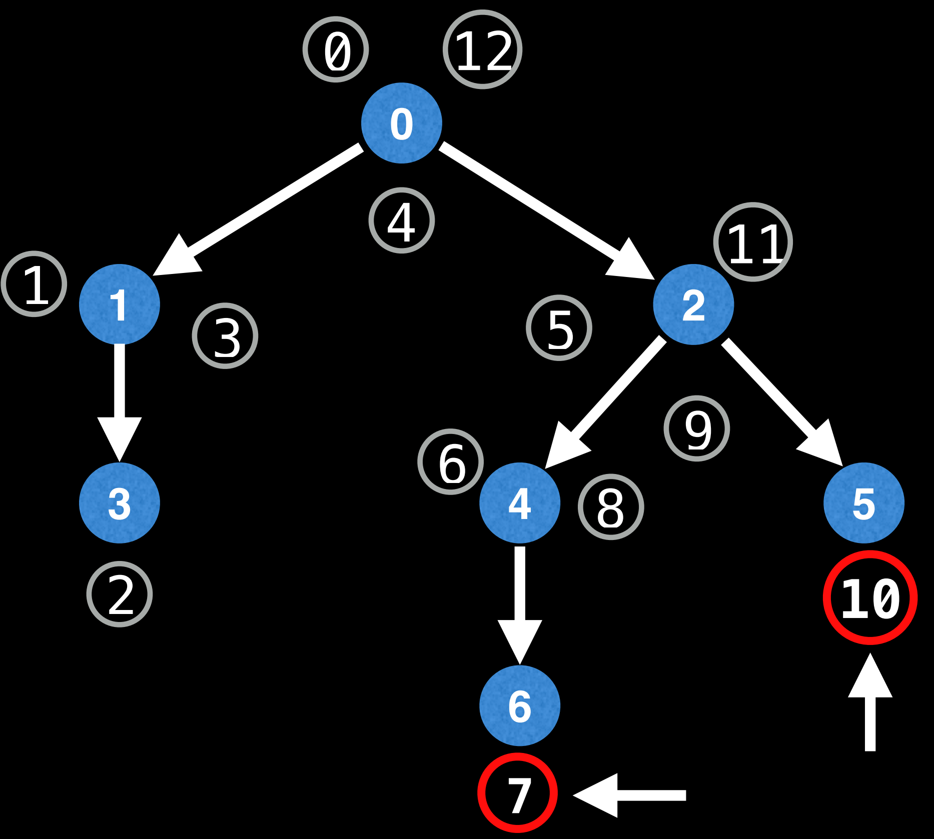
	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5	2	0



Q: What is **LCA**(6, 5)?

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5	2	0



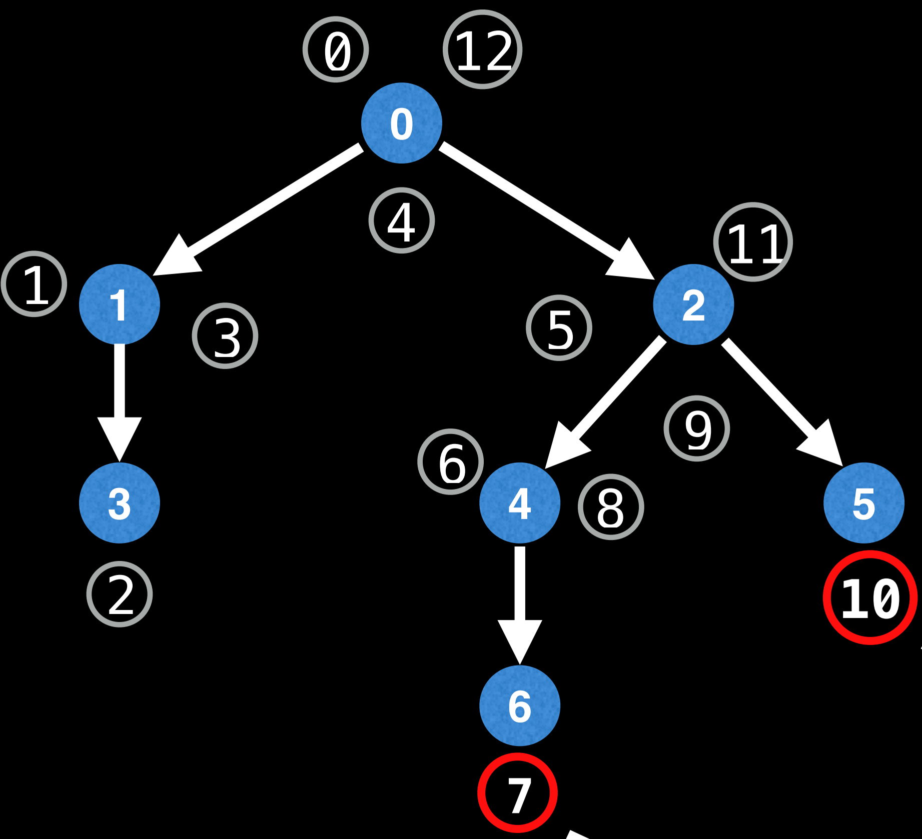
Q: What is **LCA**(6, 5)?

1. Find the index position value for the nodes `a` and `b` (5 and 6 respectably)

Nodes 5 and 6 map the the index positions 7 and 10 in the Euler Tour

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5	2	0



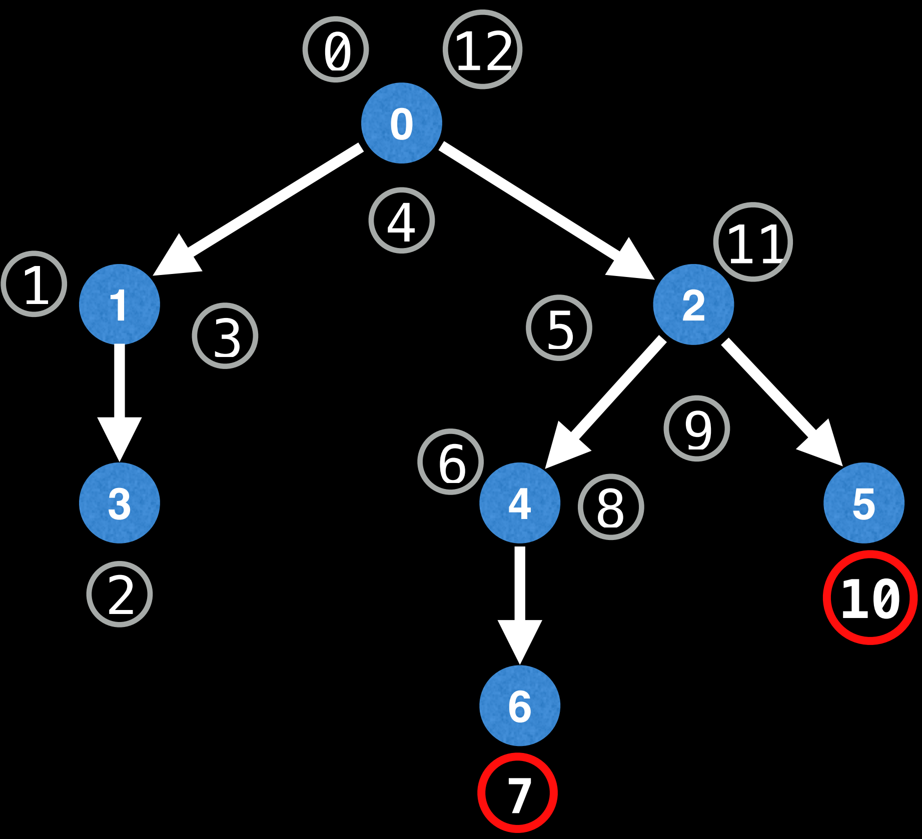
Q: What is **LCA**(6, 5)?

1. Find the index position value for the nodes `a` and `b` (5 and 6 respectably)

2. Using the depth array, find the index of the minimum value in the range of the indices obtained in step 1

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5	2	0



Q: What is **LCA**(6, 5)?

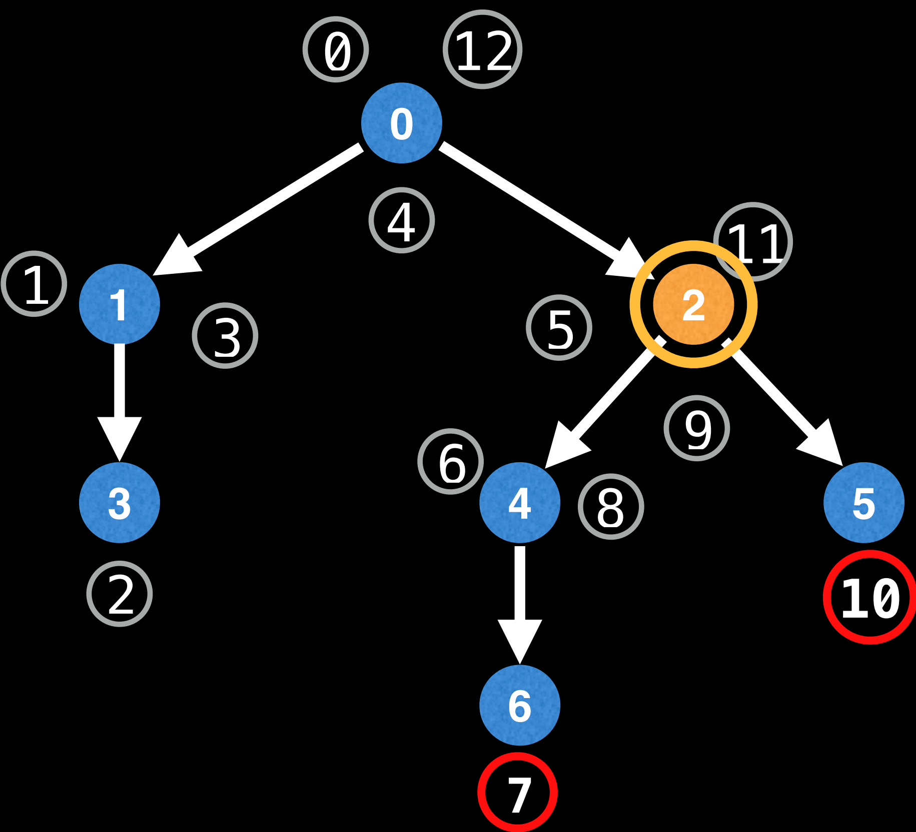
1. Find the index position value for the nodes `a` and `b` (5 and 6 respectably)

2. Using the depth array, find the index of the minimum value in the range of the indices obtained in step 1

Query the range [7, 10] in the depth array to find the index of the minimum value. This can be done in **O(1)** with a Sparse Table. For this example, the index is `9` with a value of `1`

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5	2	0



Q: What is **LCA**(6, 5)?

1. Find the index position value for the nodes `a` and `b` (5 and 6 respectably)

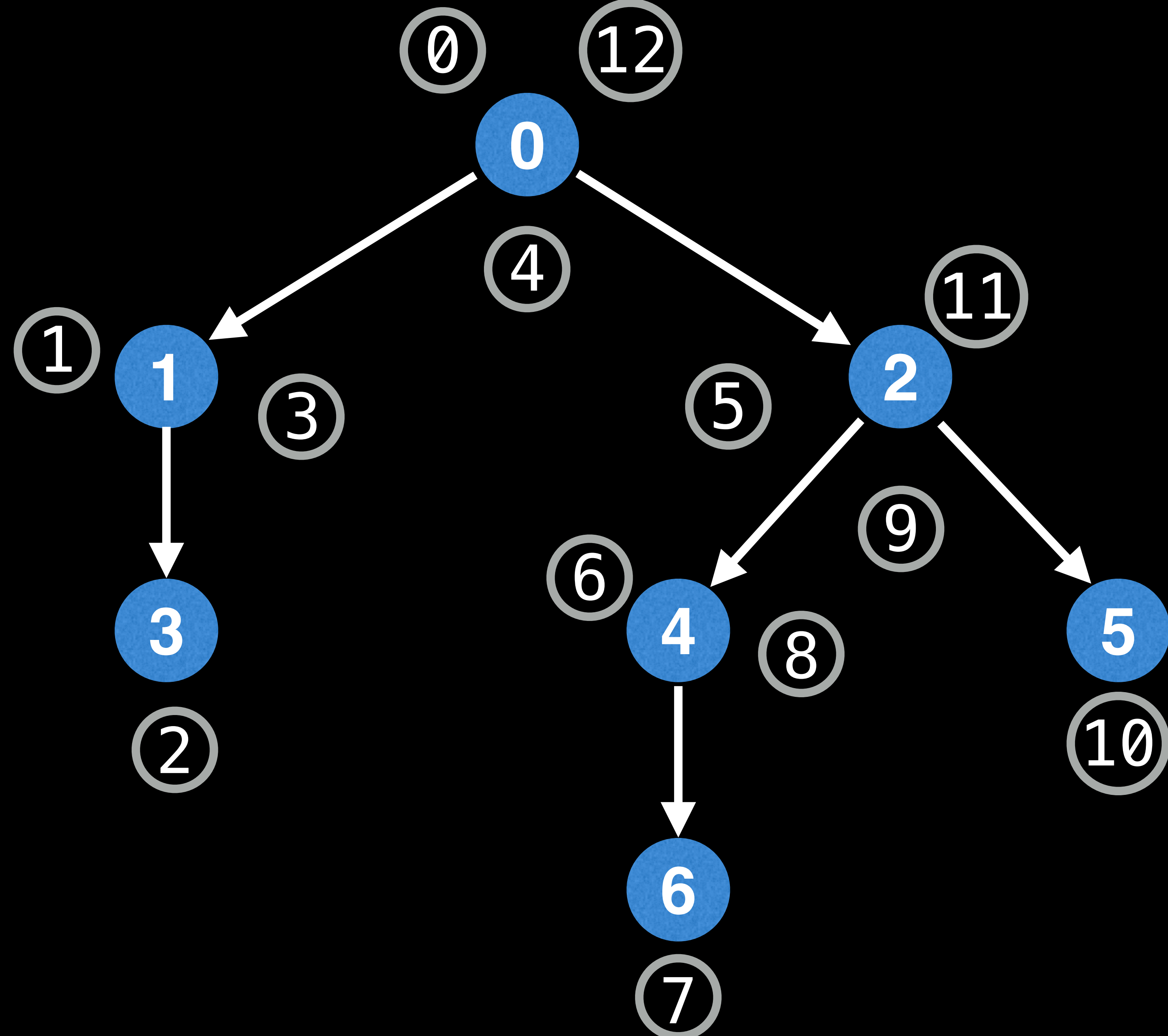
2. Using the depth array, find the index of the minimum value in the range of the indices obtained in step 1

3. Using the index obtained in step 2, find the LCA of `a` and `b` in the `nodes` array.

With index 9 found in the previous step, retrieve the LCA at nodes[9]

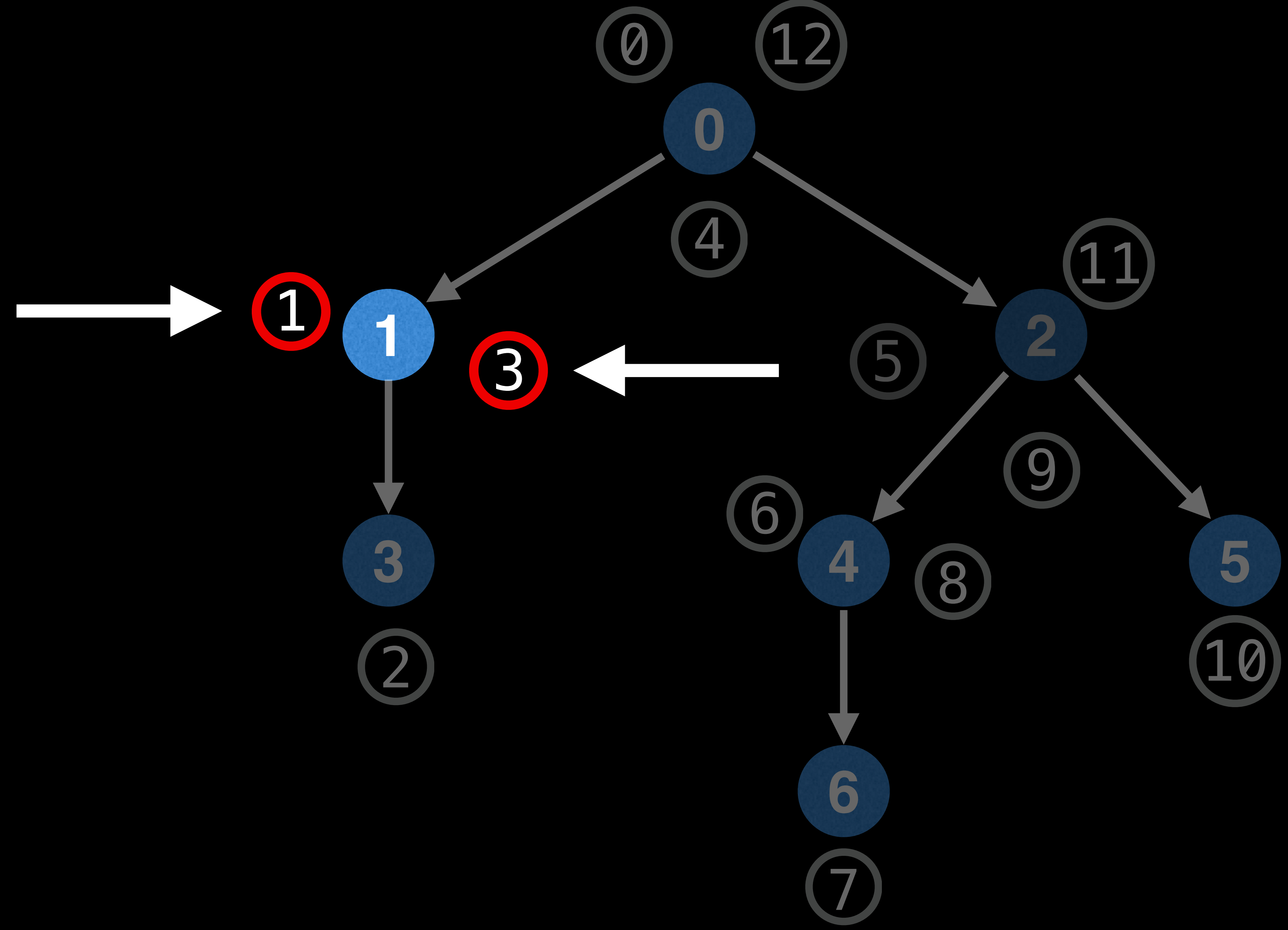
	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5	2	0

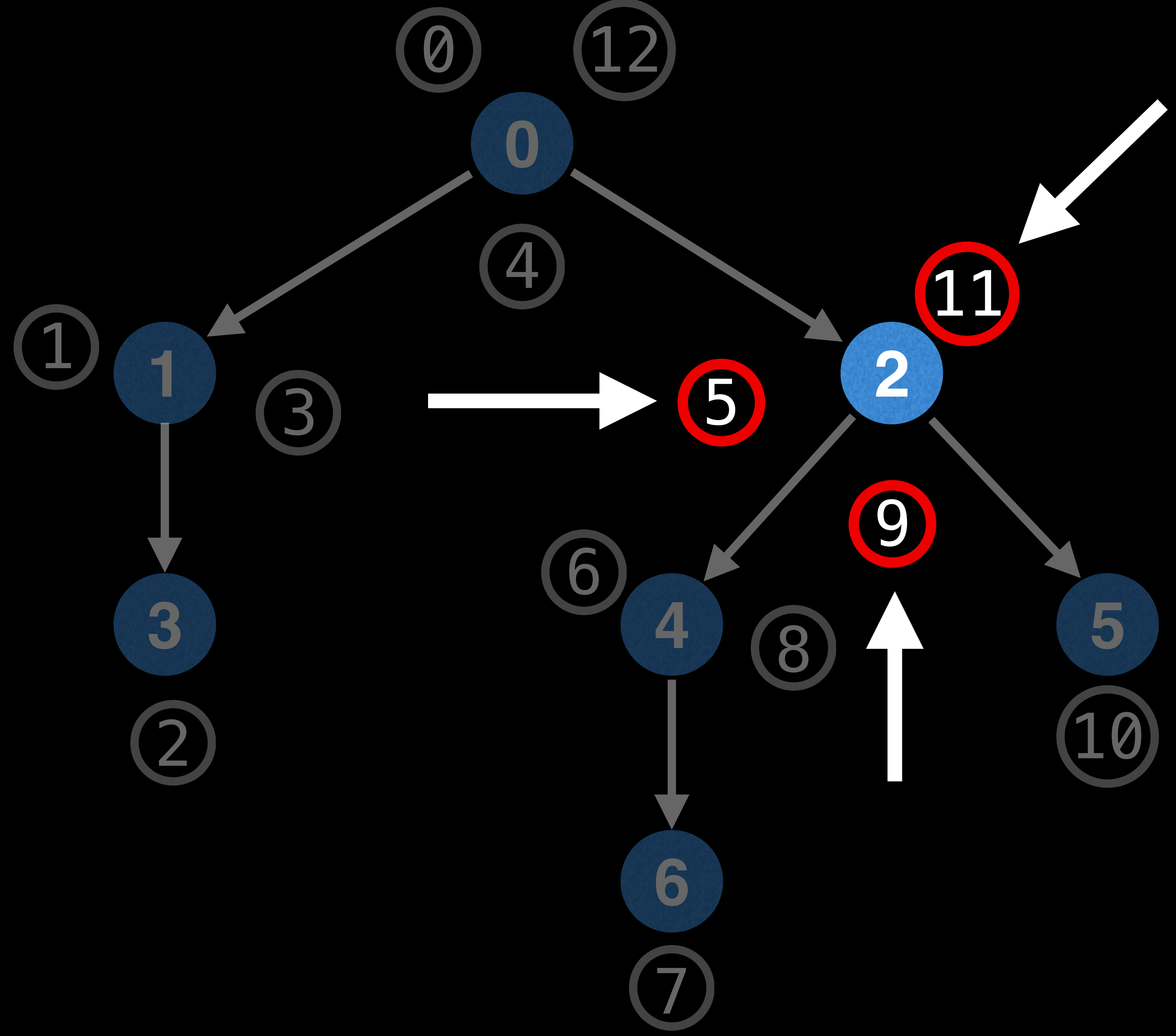


If you recall, step 1 required finding the index position for the two nodes with ids `a` and `b`.

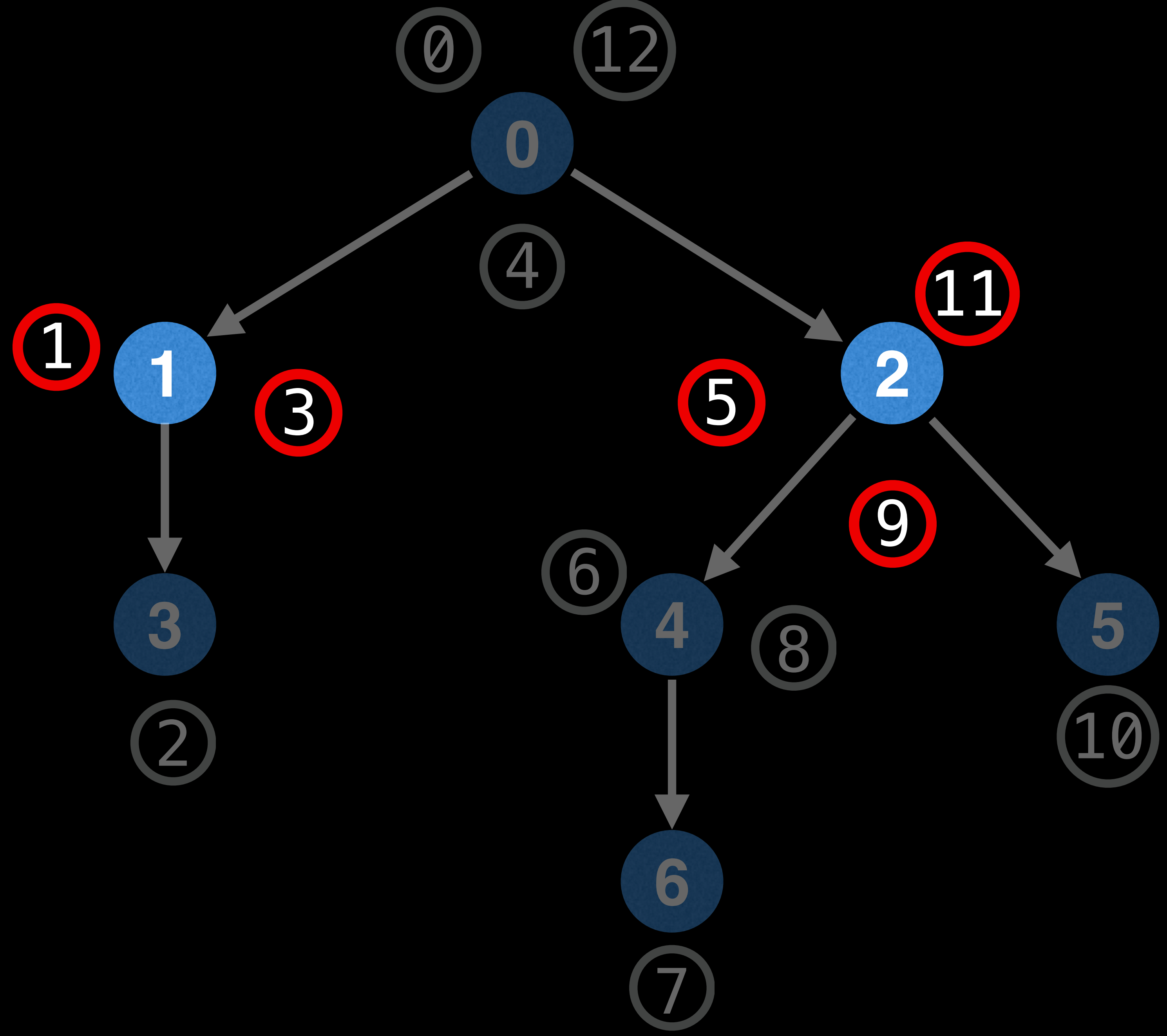
However, an issue we soon run into is that there are 2n - 1 nodes index positions in the Euler tour, and only n nodes in total, so a perfect 1 to 1 inverse mapping isn't possible.



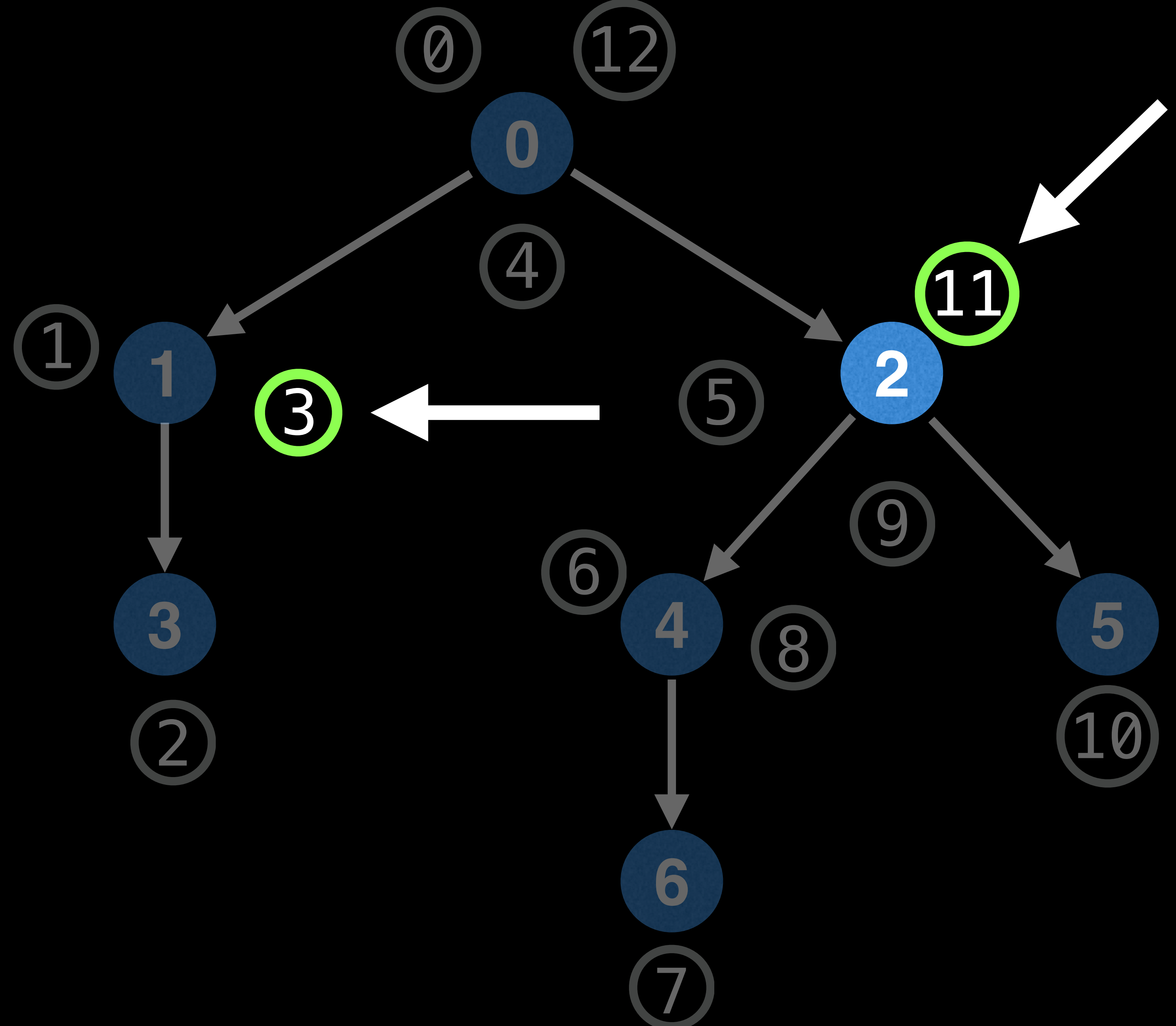
For example, the inverse mapping of node 1 could map to either index 1 or index 3 in the Euler tour.



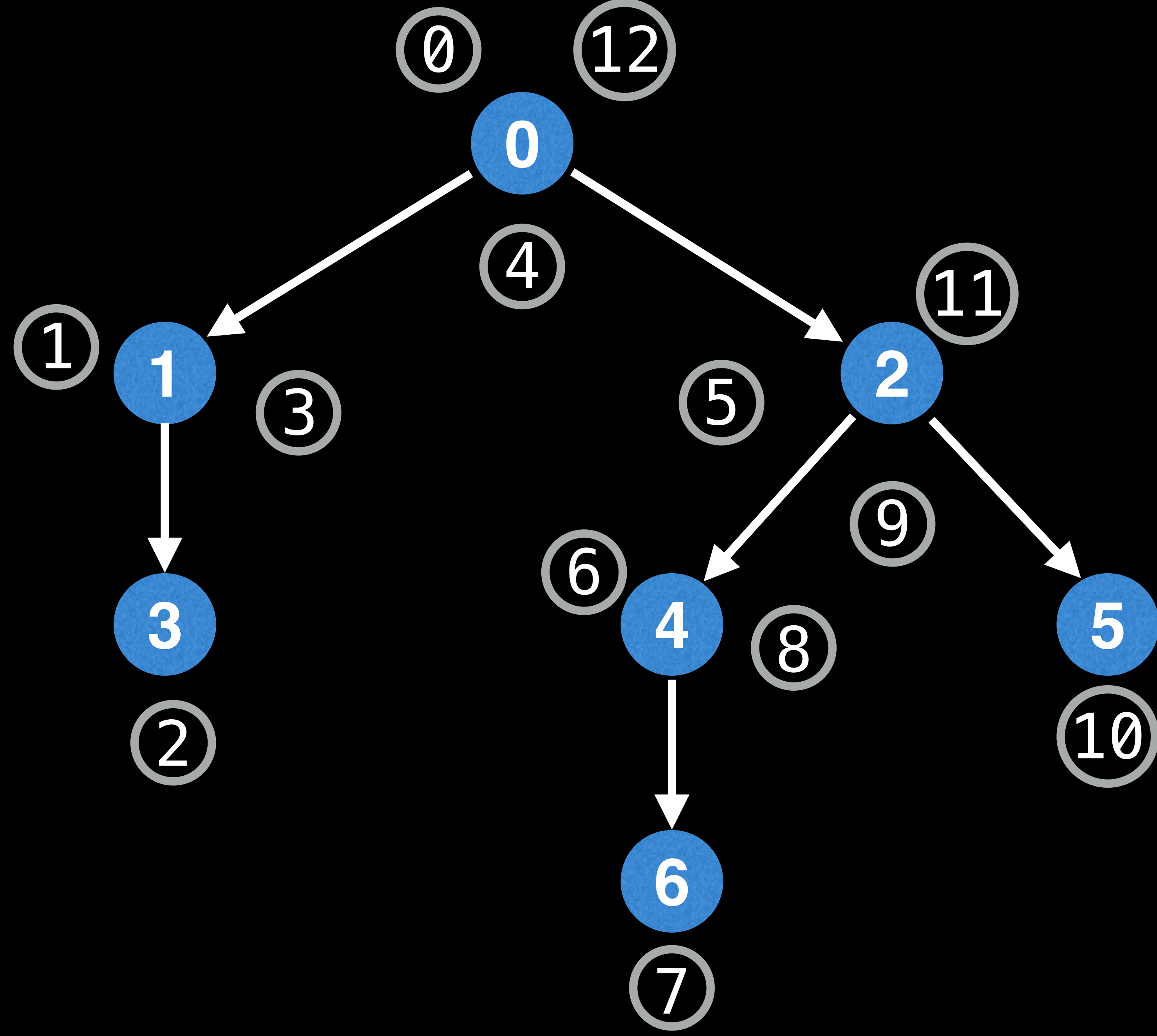
Similarly, the inverse mapping for node 2 could map to either index 5, 9 or 11 in the Euler tour.



So, which index values should we pick if we wanted to find the LCA of the nodes 1 and 2?

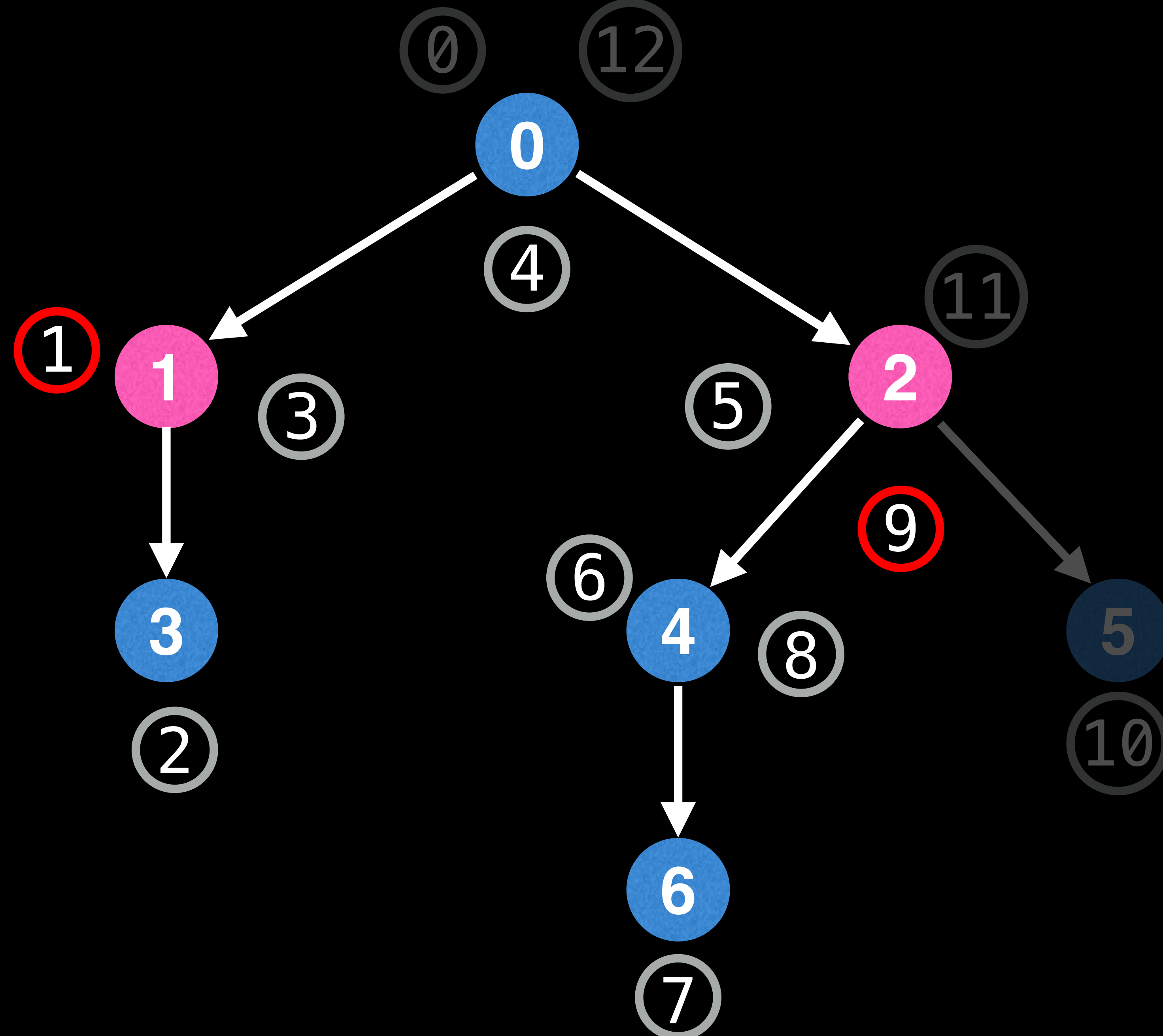


The answer is that it doesn't matter, any of the inverse index values will do. However, in practice, I find that it is easiest to select the **last encountered index** while doing the Euler tour.



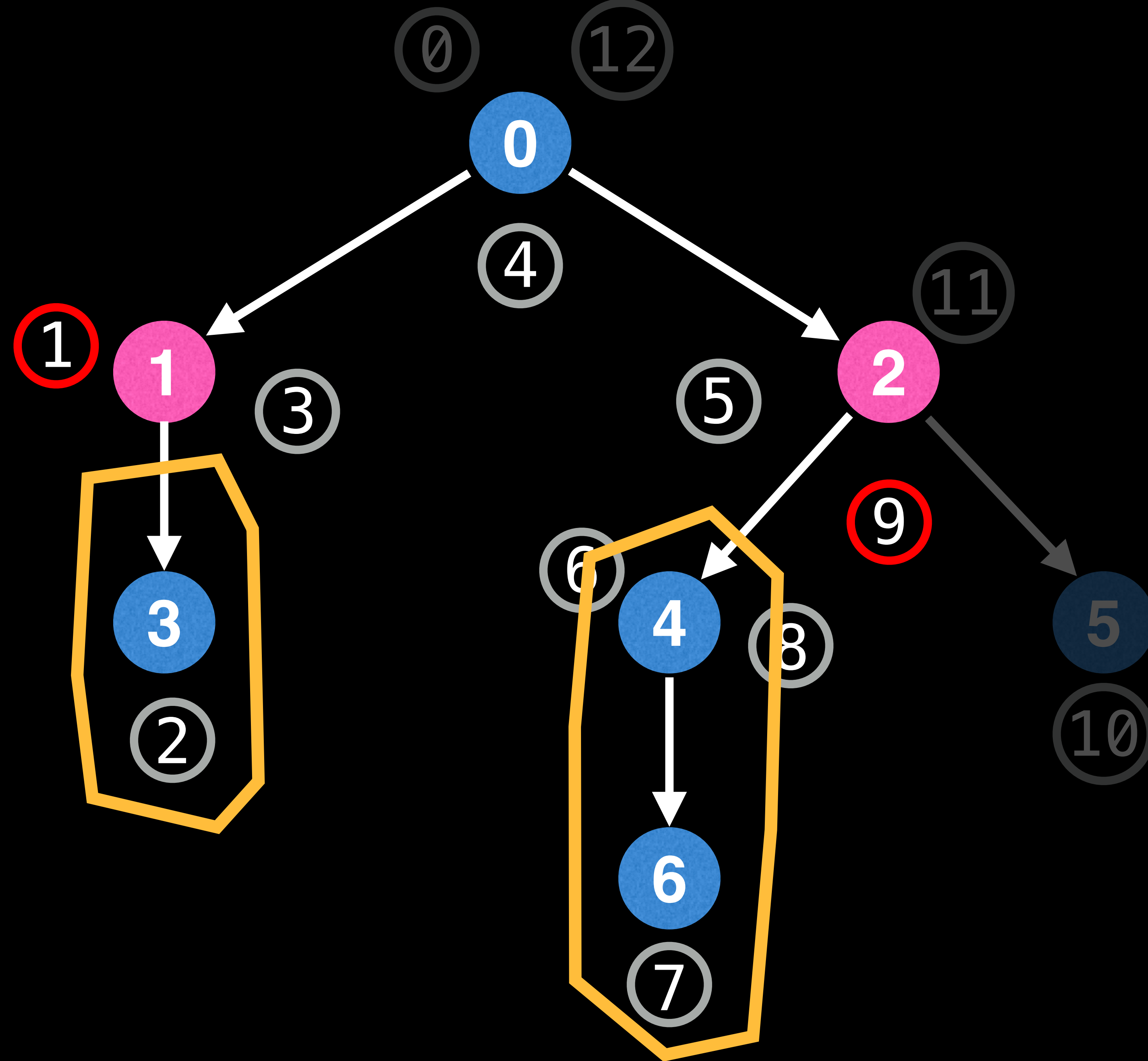
The reason the selection of the inverse index mapping doesn’t matter is that it does not affect the value obtained from the **Range Minimum Query (RMQ)** in step 2

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0



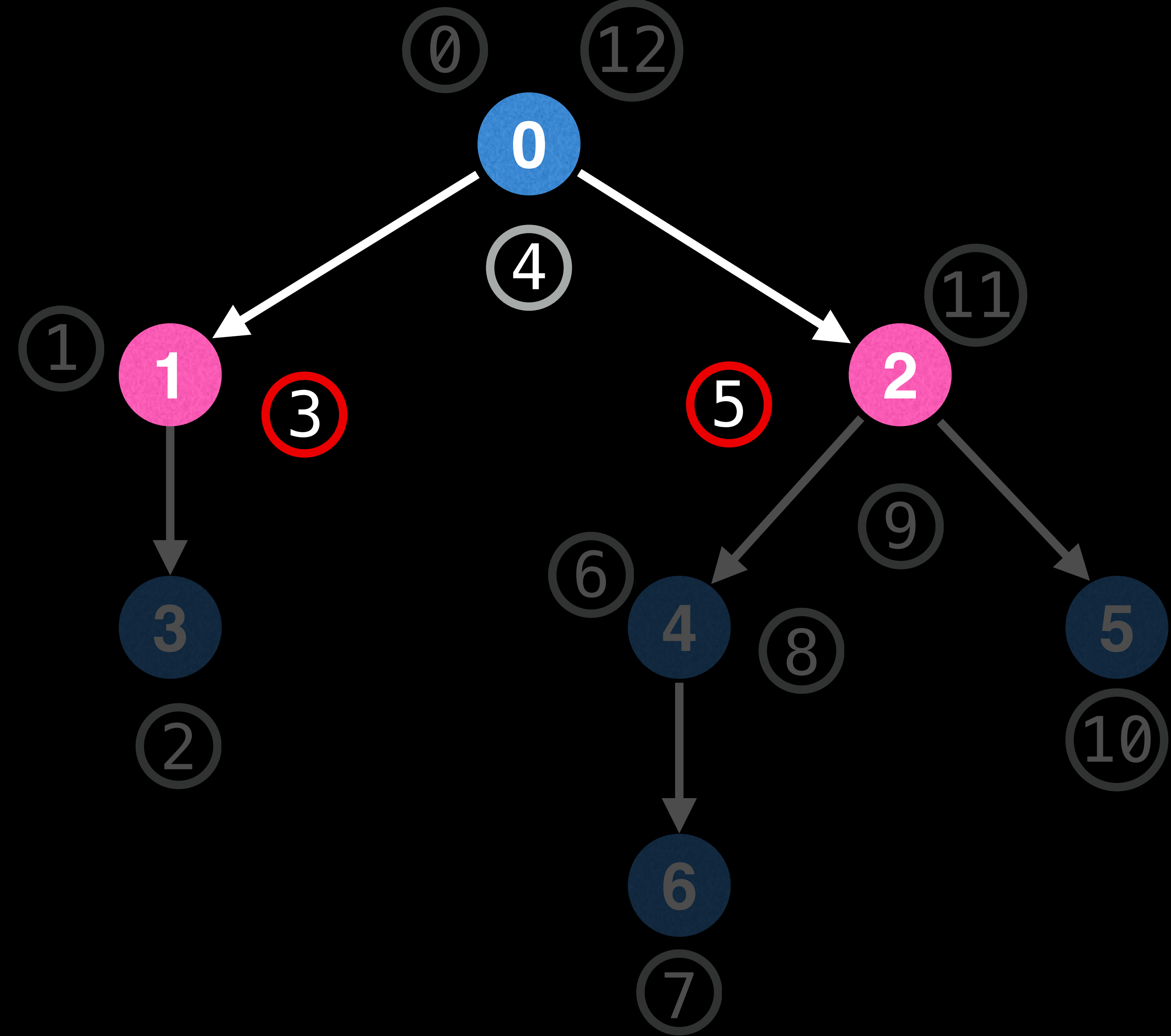
Suppose that for the LCA(1, 2) we selected index 1 for node 1 and index 9 for node 2, meaning the range [1, 9] in the depth array for the RMQ.

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0



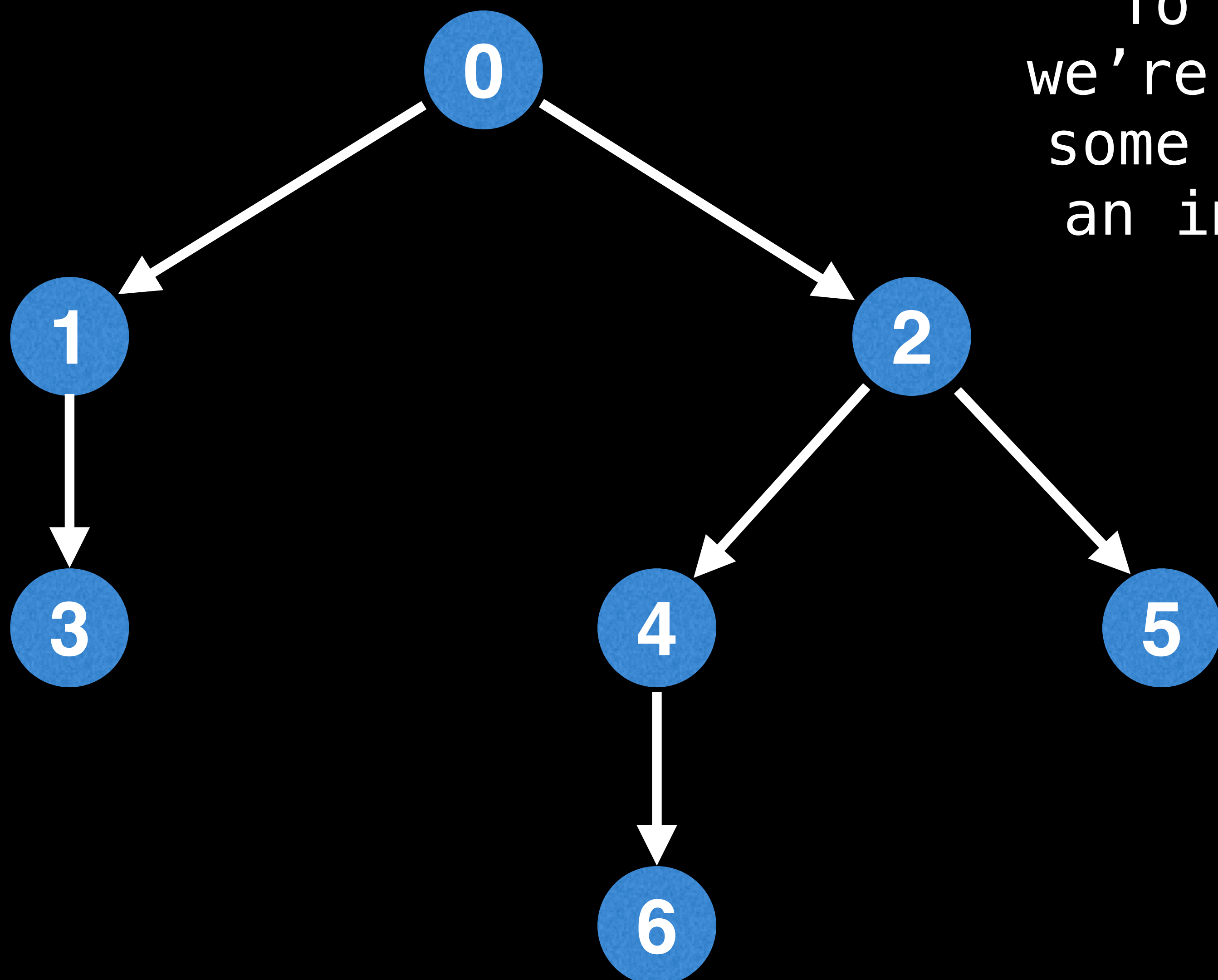
Even though the range [1, 9] includes some subtrees of the nodes 1 and 2, the depths of the subtree nodes are always more than the depths of nodes 1 and 2, so the value of the RMQ remains unchanged.

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0



You may think that choosing the index values 3 and 5 for nodes 1 and 2 would be better choice since the interval [3, 5] is smaller. However, this doesn't matter since RMQs take **O(1)** when using a sparse table.

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0

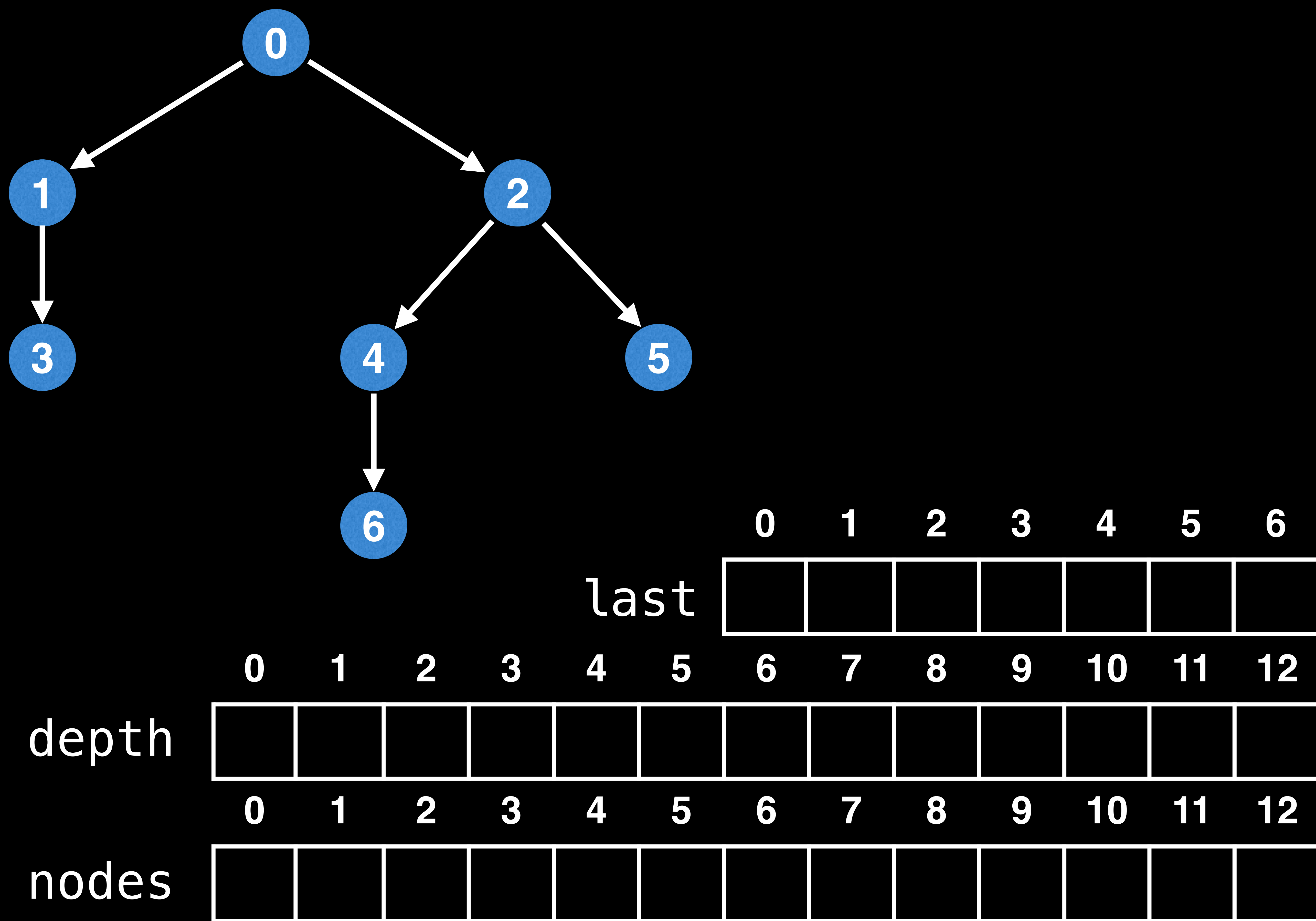


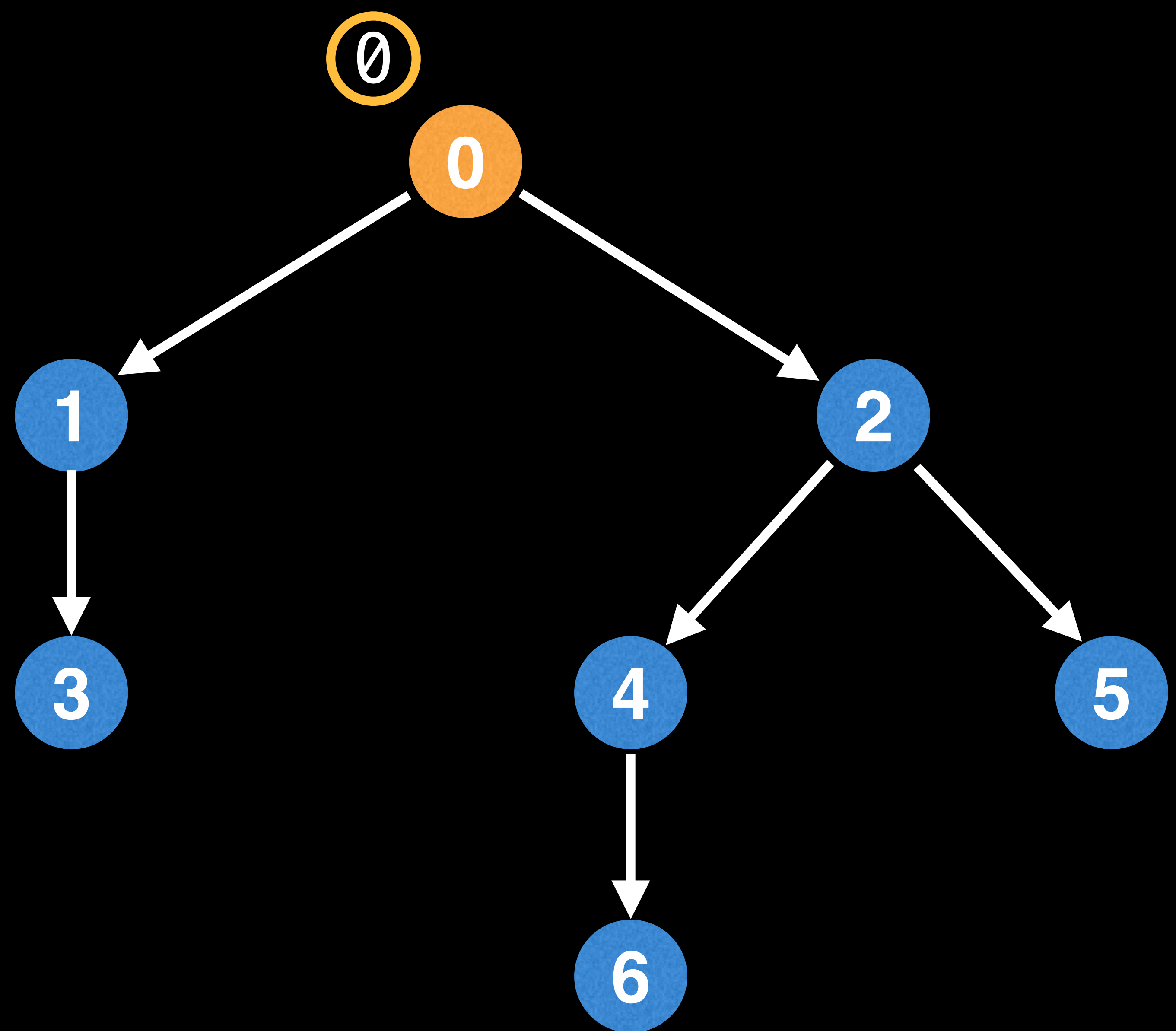
To maintain an inverse mapping, we're going to need to keep track of some additional information, namely an inverse map I will call `last`.

	0	1	2	3	4	5	6
last							

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth													

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes													

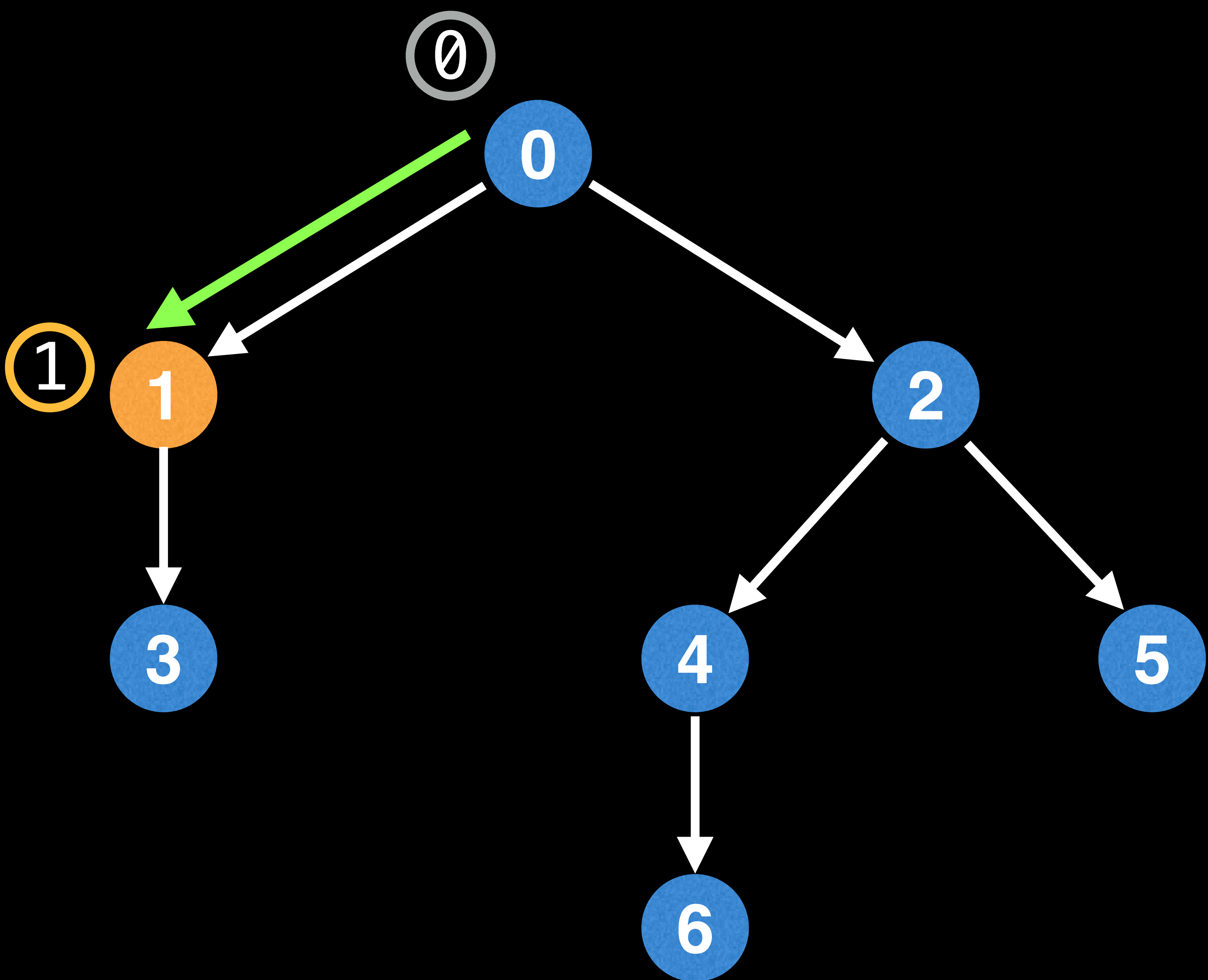

0
1
2
3
4
5
6
last
0 1 2 3 4 5 6
depth
0 1 2 3 4 5 6 7 8 9 10 11 12
nodes
0 1 2 3 4 5 6 7 8 9 10 11 12



	0	1	2	3	4	5	6
last	0						

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0												

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0												



last

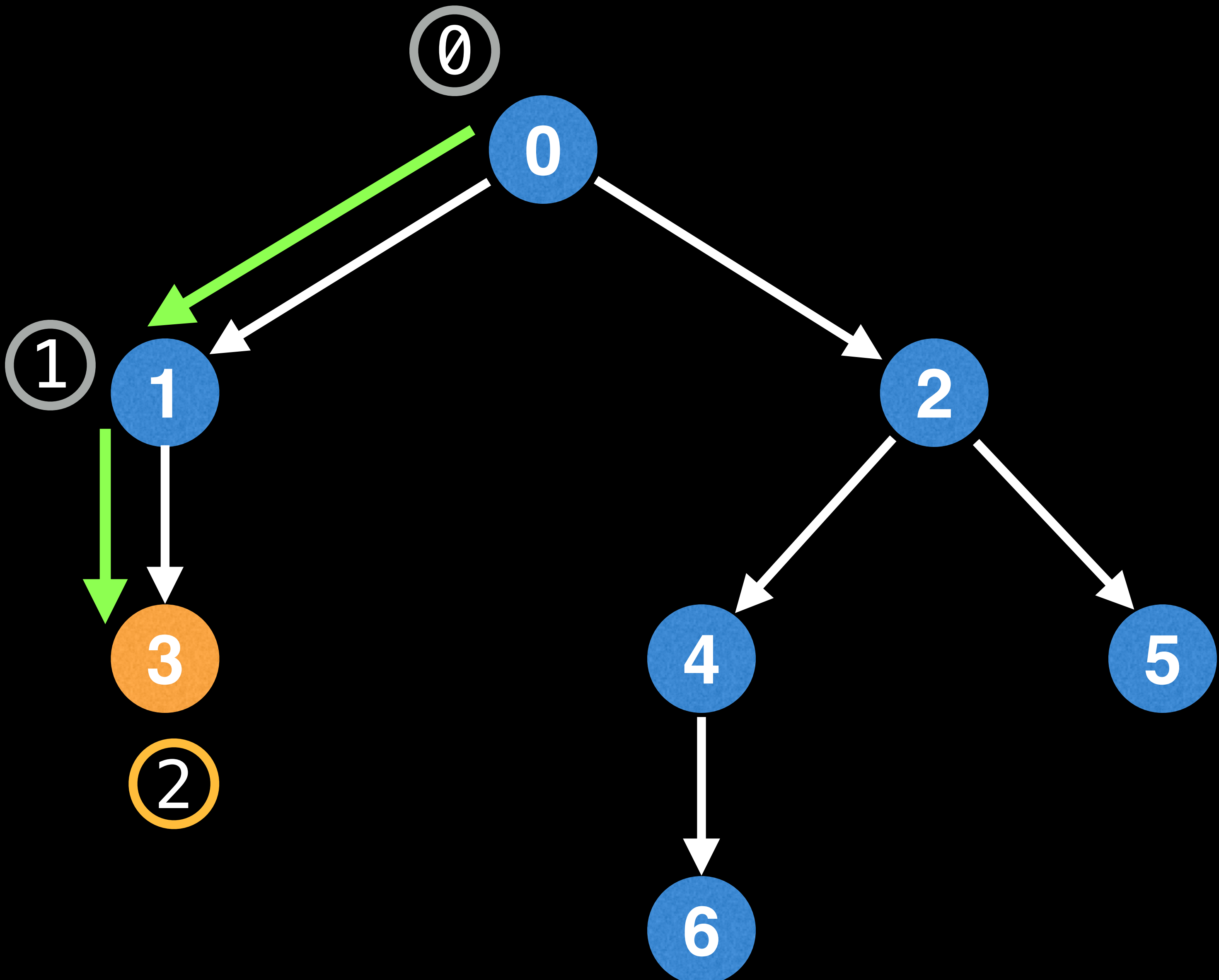
0	1	2	3	4	5	6
0	1					

depth

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1											

nodes

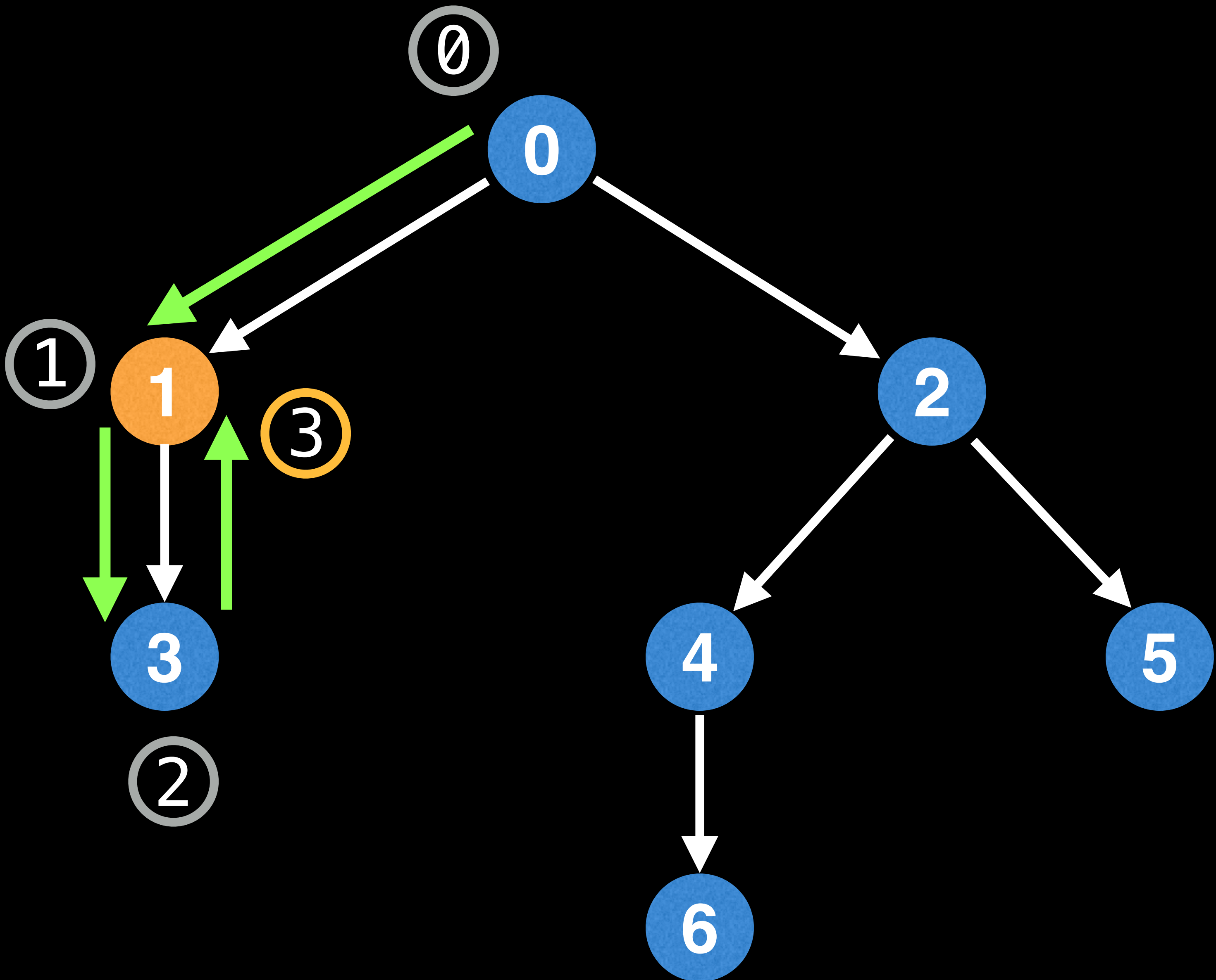
0	1	2	3	4	5	6	7	8	9	10	11	12
0	1											



	0	1	2	3	4	5	6
last	0	1		2			

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2										

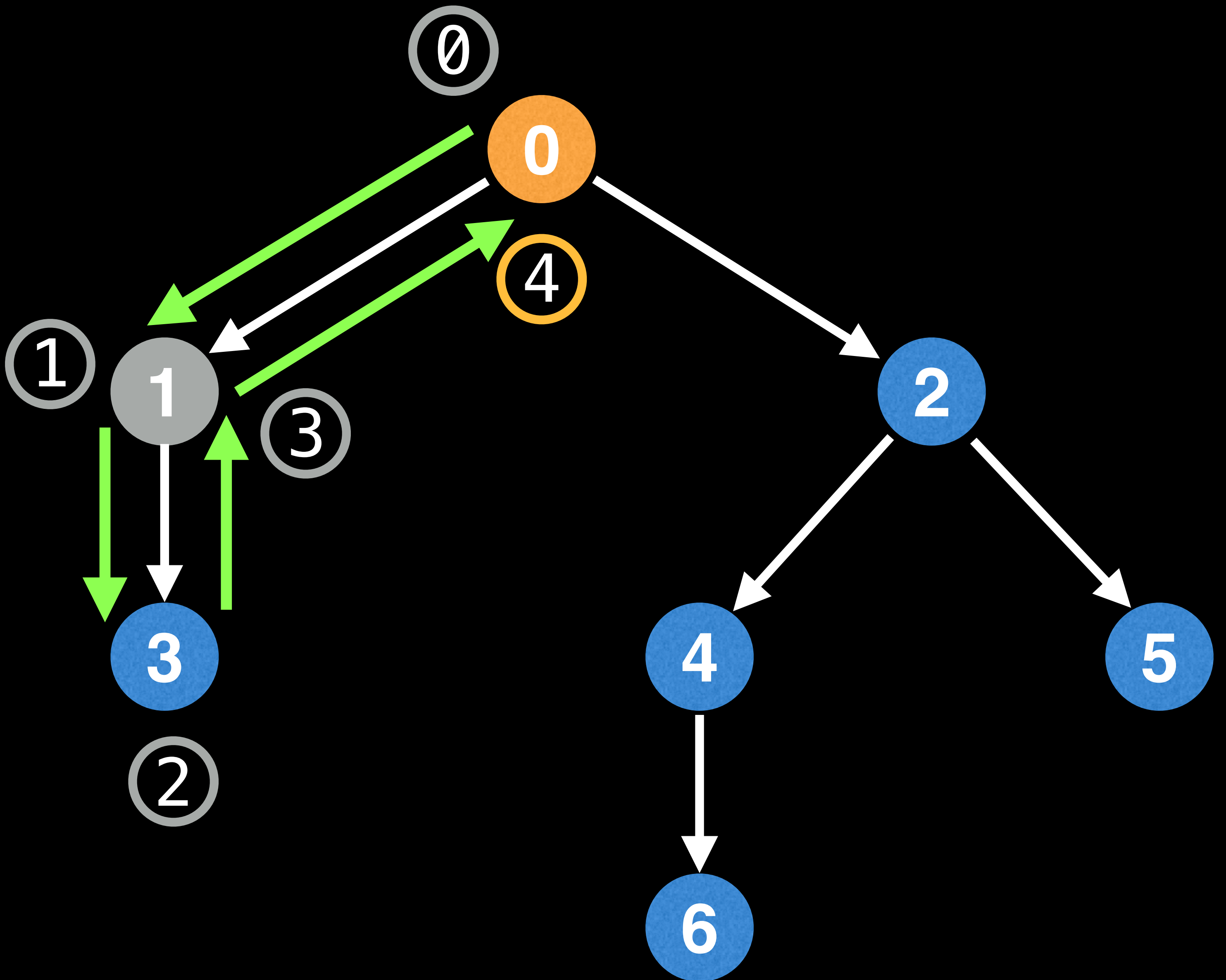
	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3										



	0	1	2	3	4	5	6
last	0	3		2			

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1									

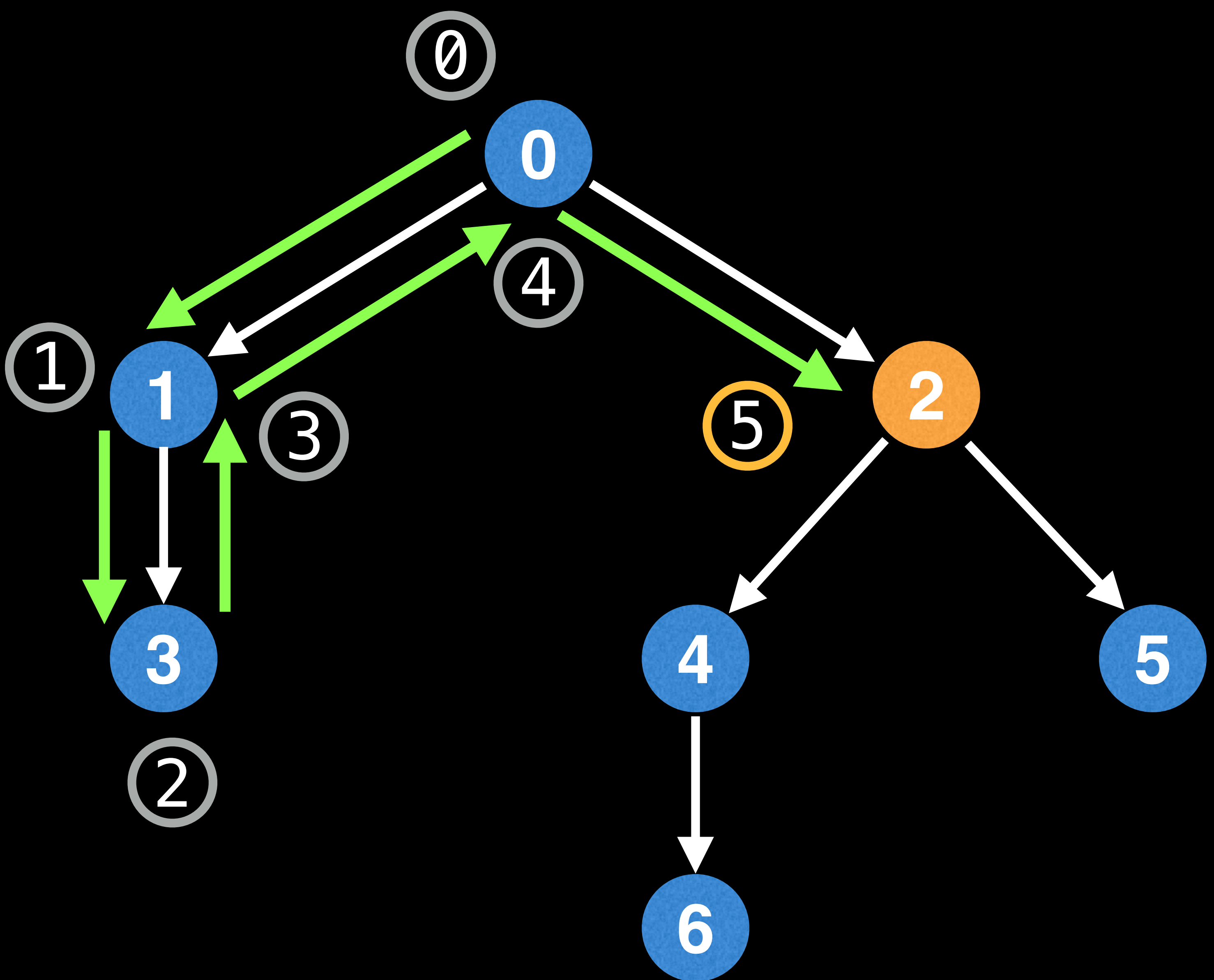
	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1									



	0	1	2	3	4	5	6
last	4	3		2			

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0								

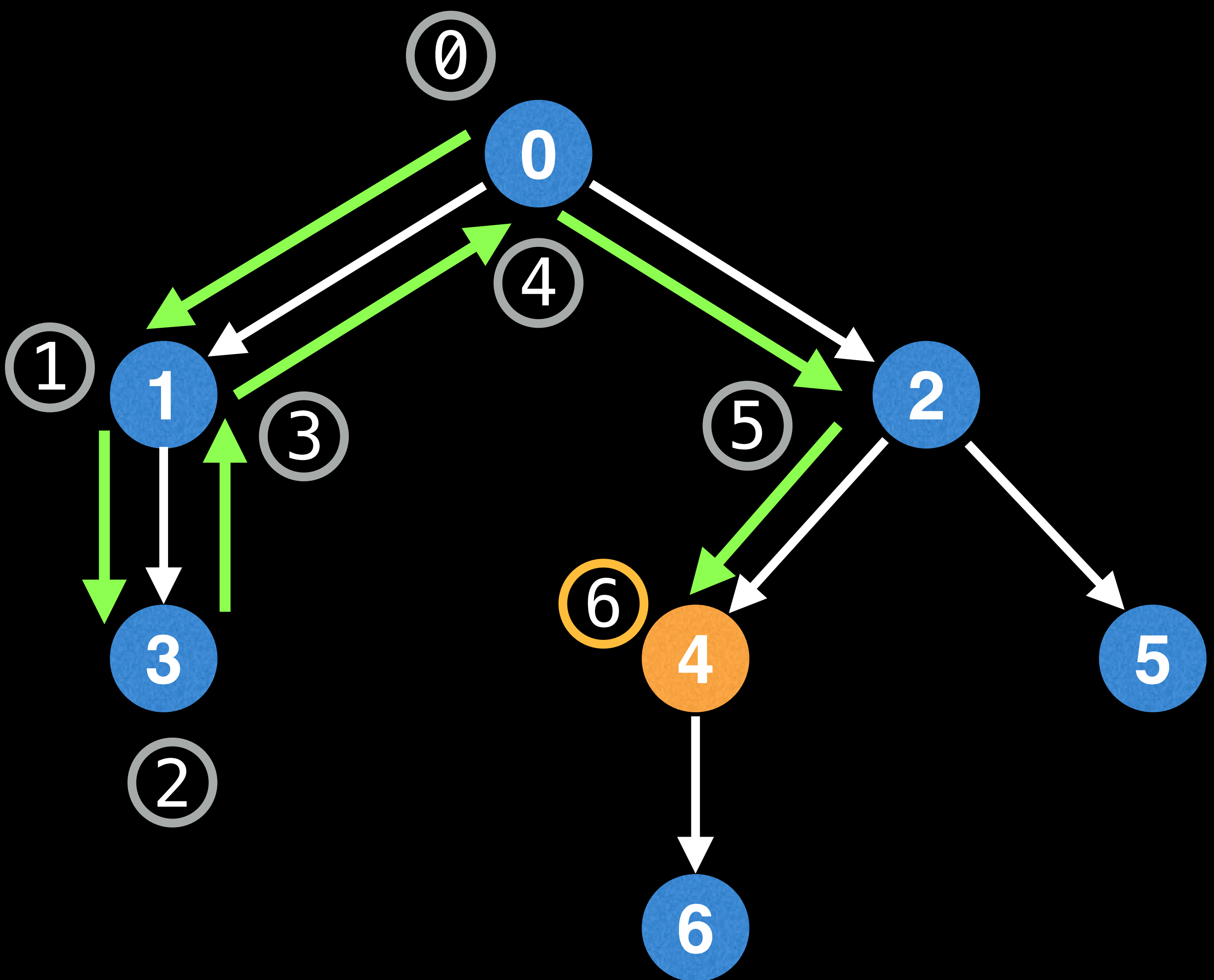
	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0								



	0	1	2	3	4	5	6
last	4	3	5	2			

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1							

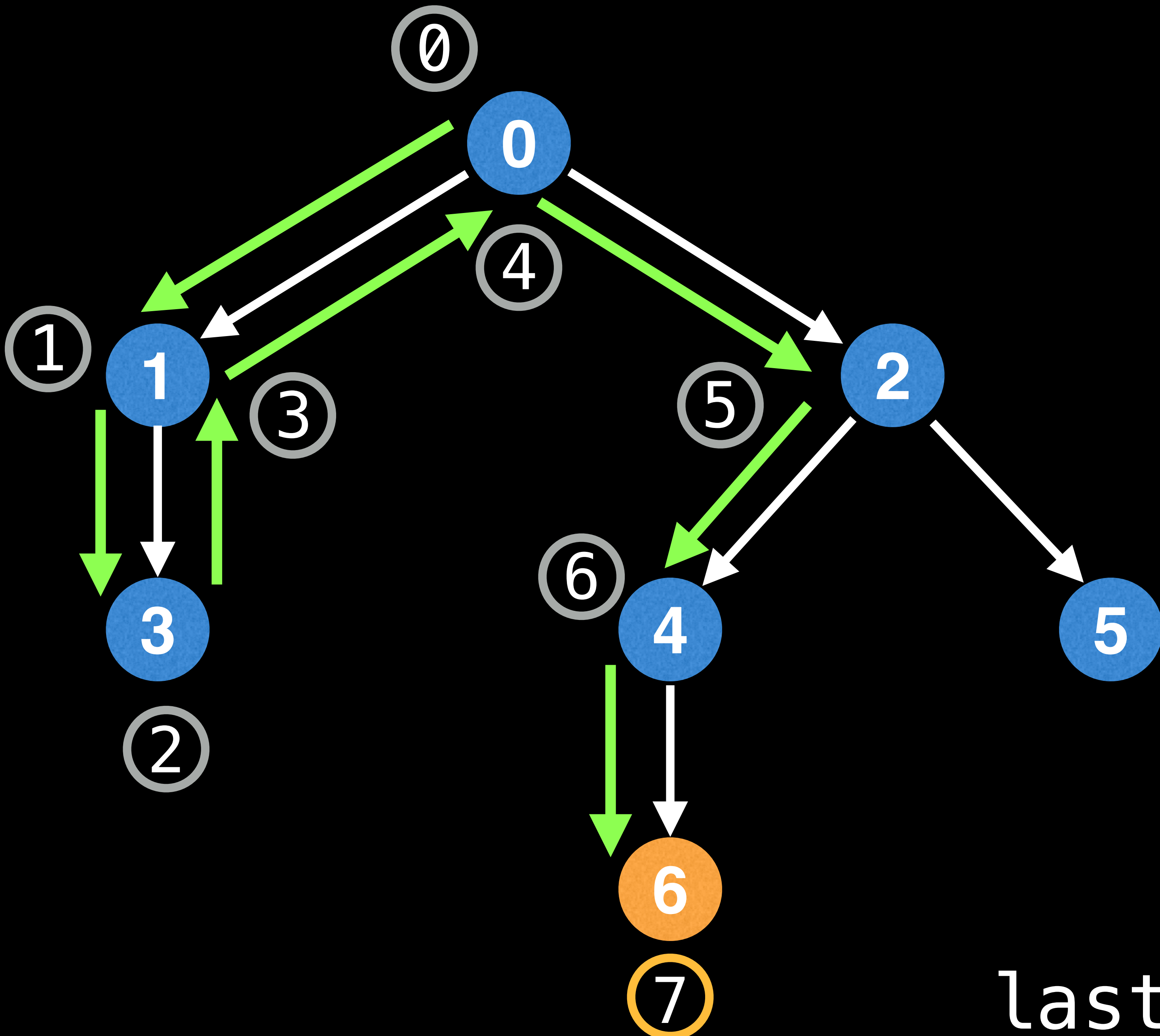
	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2							



	0	1	2	3	4	5	6
last	4	3	5	2	6		

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2						

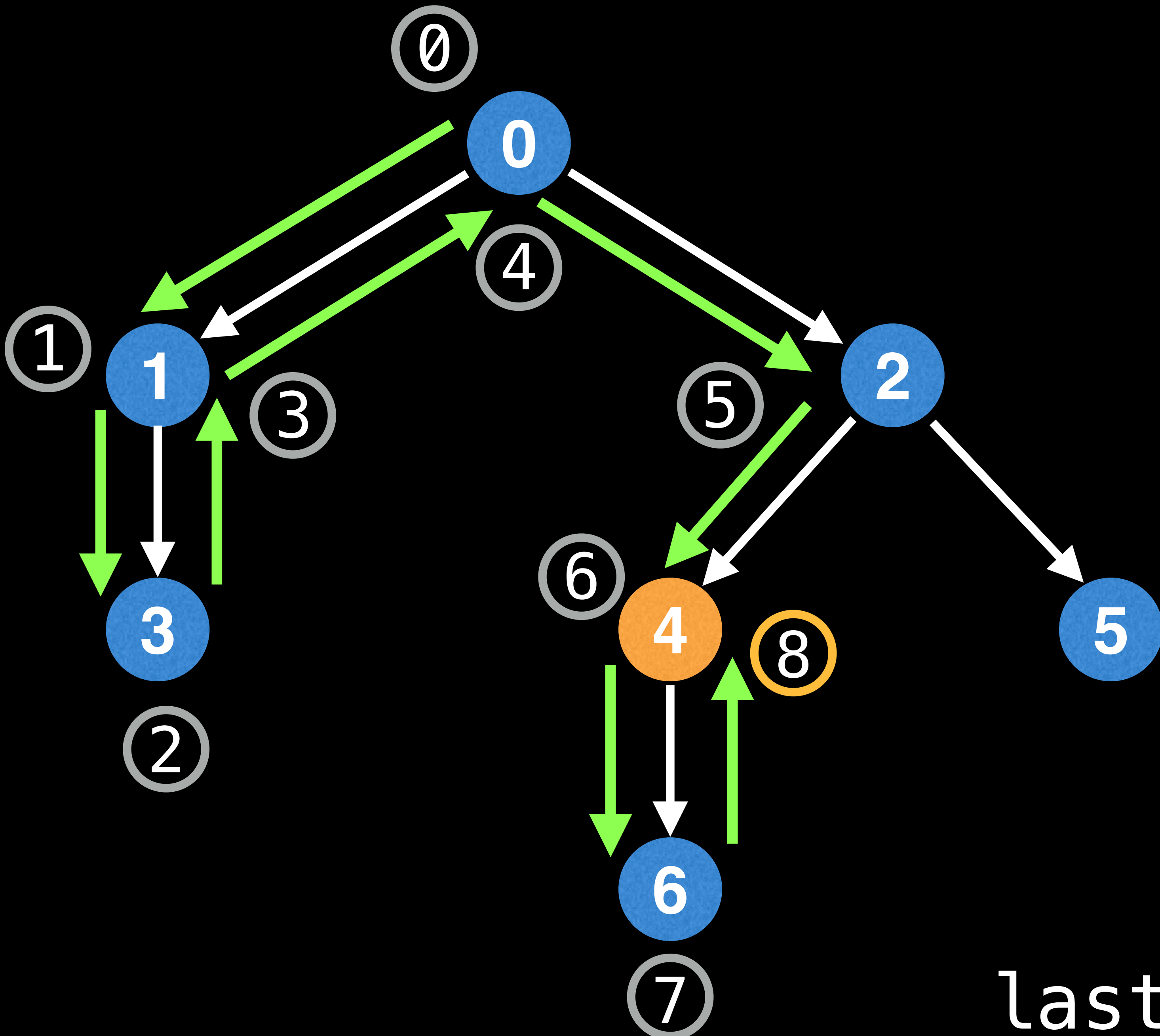
	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4						



	0	1	2	3	4	5	6
last	4	3	5	2	6		7

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3					

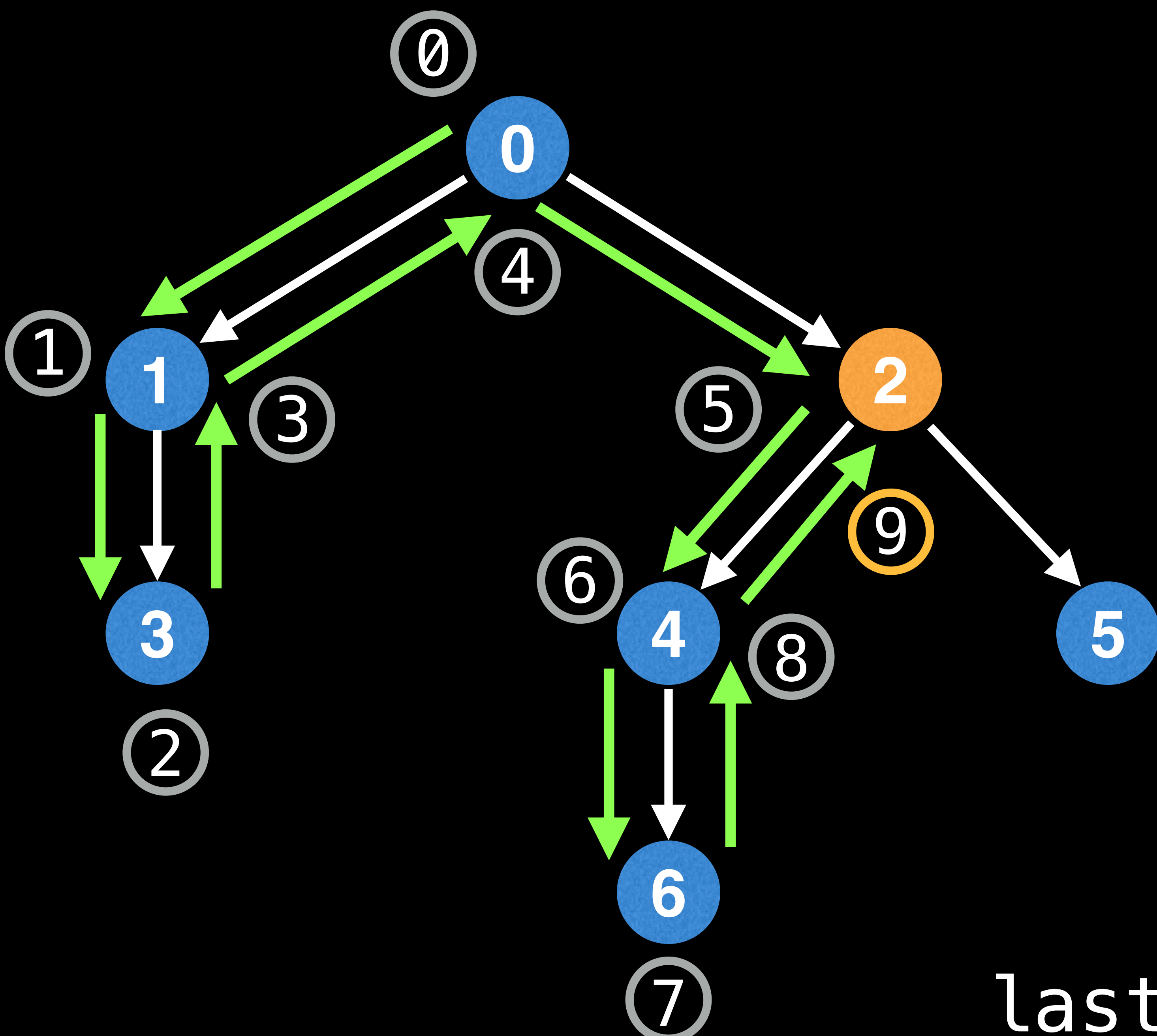
	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6					



	0	1	2	3	4	5	6
last	4	3	5	2	8		7

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2				

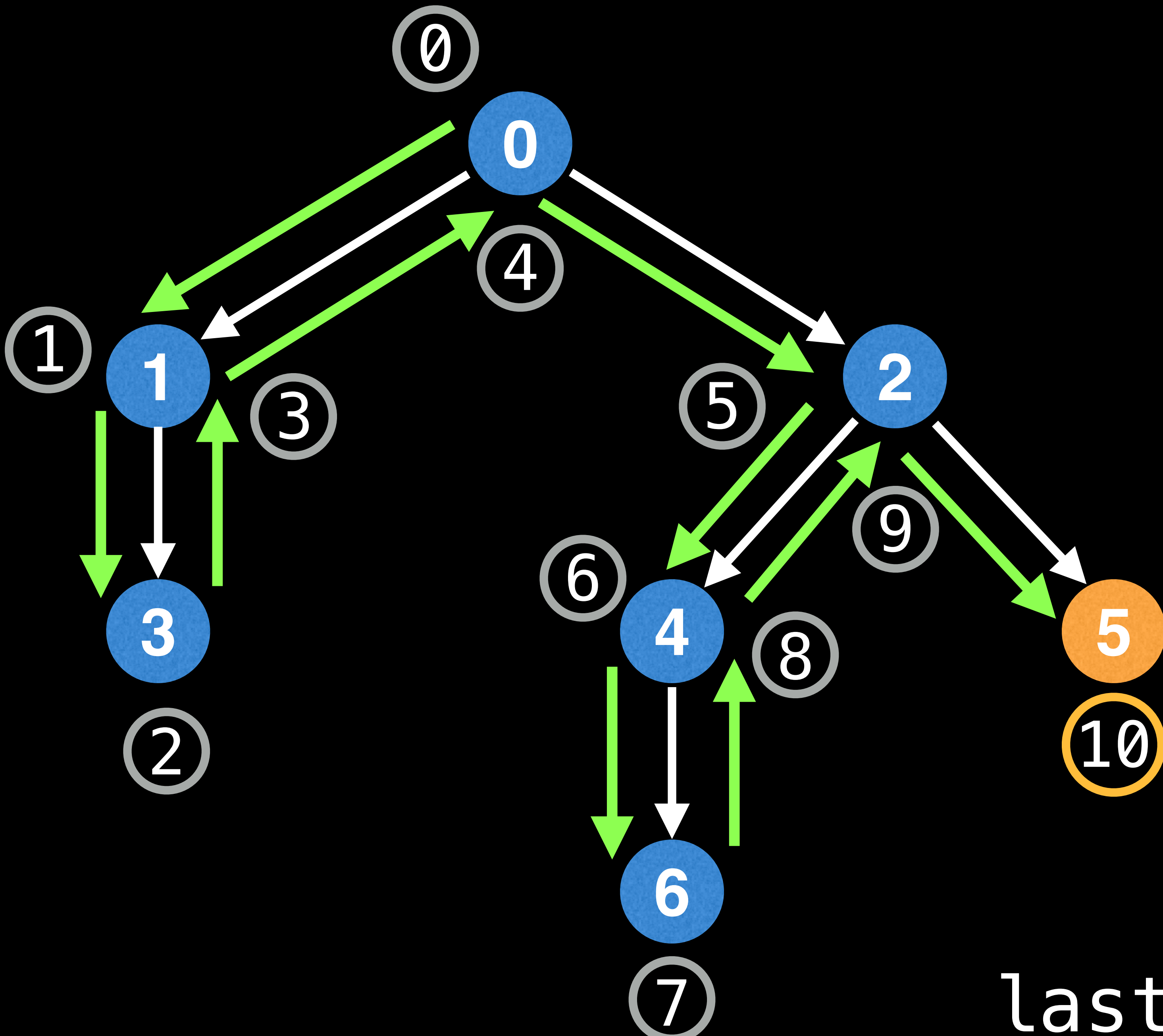
	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4				



	0	1	2	3	4	5	6
last	4	3	9	2	8		7

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1			

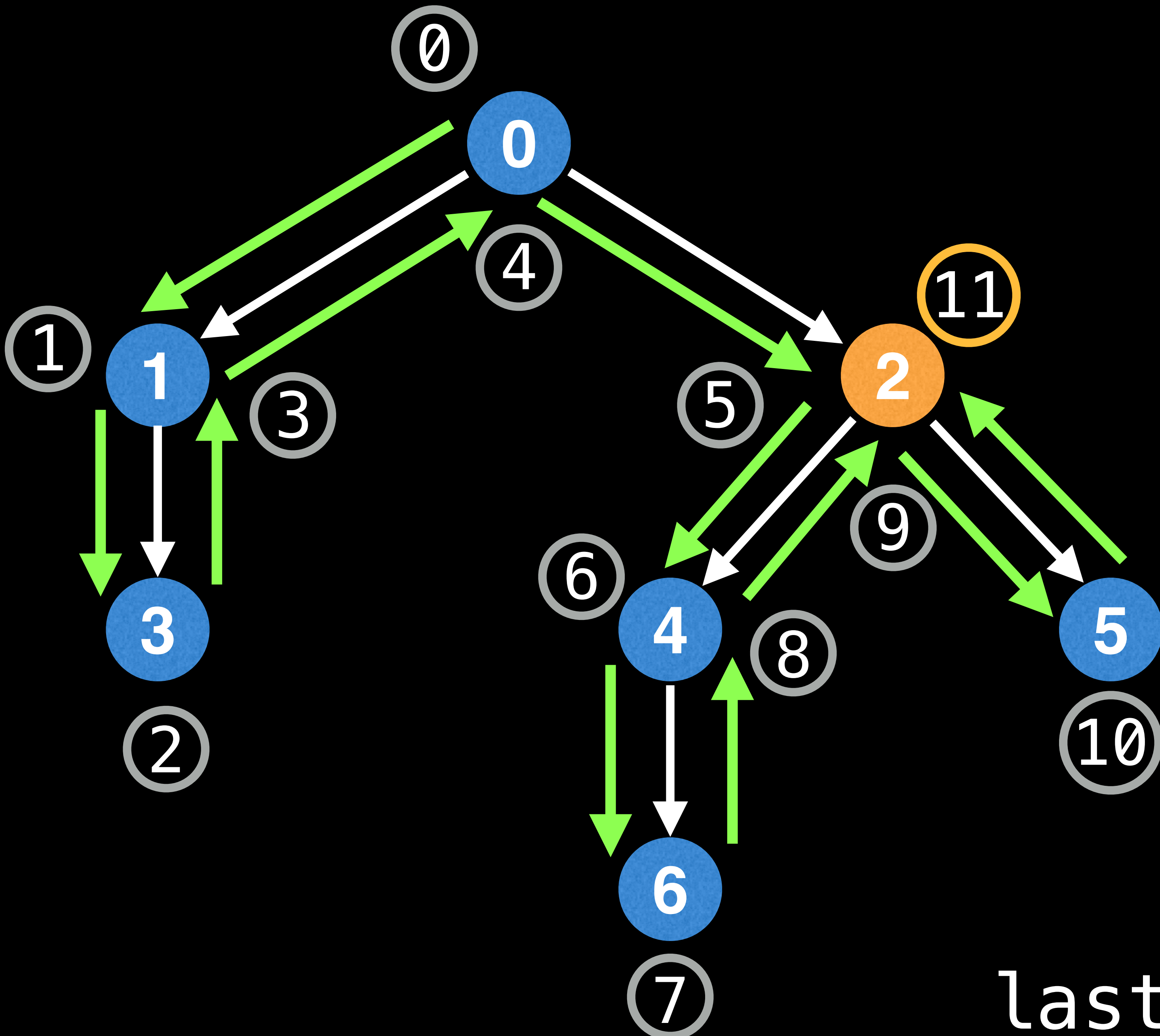
	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2			



	0	1	2	3	4	5	6
last	4	3	9	2	8	10	7

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2		

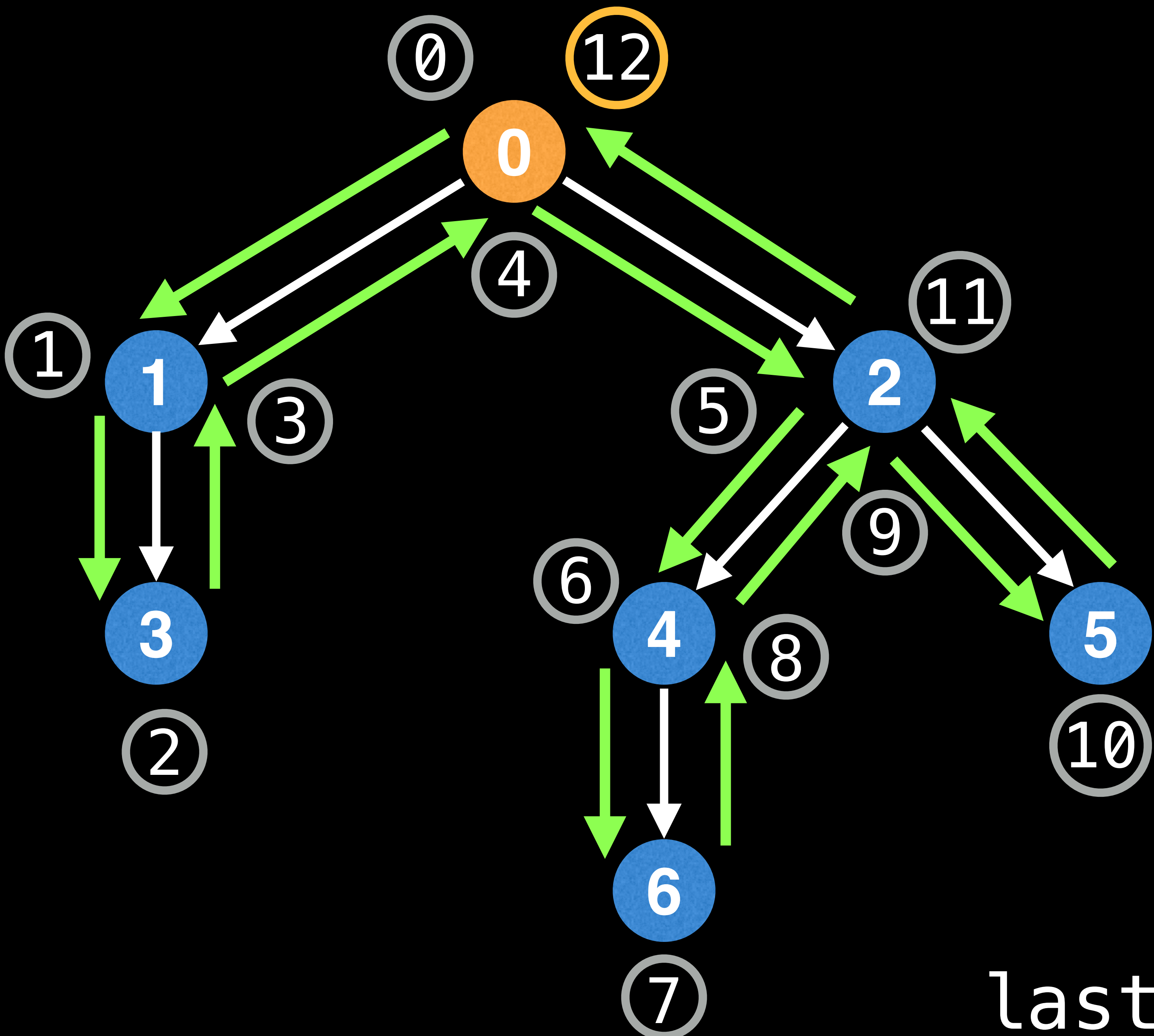
	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5		



	0	1	2	3	4	5	6
last	4	3	11	2	8	10	7

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	

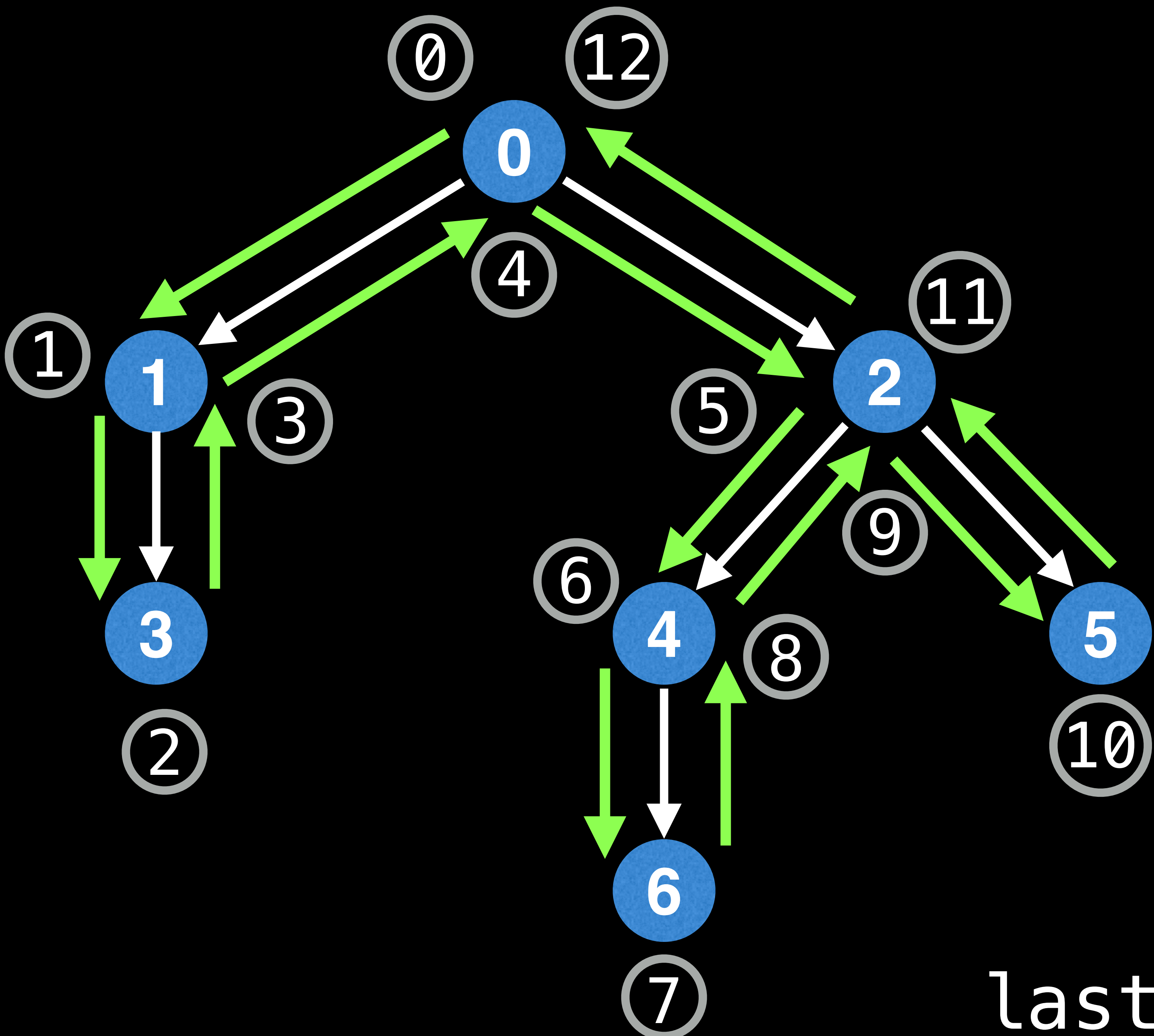
	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5	2	



	0	1	2	3	4	5	6
last	12	3	11	2	8	10	7

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5	2	0



	0	1	2	3	4	5	6
last	12	3	11	2	8	10	7

	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0

	0	1	2	3	4	5	6	7	8	9	10	11	12
nodes	0	1	3	1	0	2	4	6	4	2	5	2	0

```
class TreeNode:

  # A unique index (id) associated with this
  # TreeNode.
  int index;

  # List of pointers to child TreeNodes.
  TreeNode children[];
```

```
class TreeNode:

  # A unique index (id) associated with this
  # TreeNode.
  int index;

  # List of pointers to child TreeNodes.
  TreeNode children[];
```

```
class TreeNode:

  # A unique index (id) associated with this
  # TreeNode.
  int index;

  # List of pointers to child TreeNodes.
  TreeNode children[];
```

```
function setup(n, root):

  nodes = … # array of nodes of size 2n - 1
  depth = … # array of integers of size 2n - 1

  last = … # node index -> Euler tour index

  # Do Eulerian Tour around the tree
  dfs(root)

  # Initialize sparse table data structure to
  # do Range Minimum Queries (RMQs) on the
  # `depth` array. Sparse tables take O(nlogn)
  # time to construct and do RMQs in O(1)
  sparse_table = CreateMinSparseTable(depth)
```

```
function setup(n, root):

  nodes = … # array of nodes of size 2n - 1
  depth = … # array of integers of size 2n - 1

  last = … # node index -> Euler tour index

  # Do Eulerian Tour around the tree
  dfs(root)

  # Initialize sparse table data structure to
  # do Range Minimum Queries (RMQs) on the
  # `depth` array. Sparse tables take O(nlogn)
  # time to construct and do RMQs in O(1)
  sparse_table = CreateMinSparseTable(depth)
```

```
function setup(n, root):

  nodes = … # array of nodes of size 2n – 1
  depth = … # array of integers of size 2n – 1

  last = … # node index -> Euler tour index

  # Do Eulerian Tour around the tree
  dfs(root)

  # Initialize sparse table data structure to
  # do Range Minimum Queries (RMQs) on the
  # `depth` array. Sparse tables take O(nlogn)
  # time to construct and do RMQs in O(1)
  sparse_table = CreateMinSparseTable(depth)
```

```
function setup(n, root):

  nodes = … # array of nodes of size 2n - 1
  depth = … # array of integers of size 2n - 1

  last = … # node index -> Euler tour index

  # Do Eulerian Tour around the tree
  dfs(root)

  # Initialize sparse table data structure to
  # do Range Minimum Queries (RMQs) on the
  # `depth` array. Sparse tables take O(nlogn)
  # time to construct and do RMQs in O(1)
  sparse_table = CreateMinSparseTable(depth)
```

```
function setup(n, root):

  nodes = … # array of nodes of size 2n - 1
  depth = … # array of integers of size 2n - 1

  last = … # node index -> Euler tour index

  # Do Eulerian Tour around the tree
  dfs(root)

  # Initialize sparse table data structure to
  # do Range Minimum Queries (RMQs) on the
  # `depth` array. Sparse tables take O(nlogn)
  # time to construct and do RMQs in O(1)
  sparse_table = CreateMinSparseTable(depth)
```

```
function setup(n, root):

  nodes = … # array of nodes of size 2n - 1
  depth = … # array of integers of size 2n - 1

  last = … # node index -> Euler tour index

  # Do Eulerian Tour around the tree
  dfs(root)

  # Initialize sparse table data structure to
  # do Range Minimum Queries (RMQs) on the
  # `depth` array. Sparse tables take O(nlogn)
  # time to construct and do RMQs in O(1)
  sparse_table = CreateMinSparseTable(depth)
```

```
# Eulerian tour index position
tour_index = 0

# Do an Eulerian Tour of all the nodes using
# a DFS traversal.
function dfs(node, node_depth = 0):
  if node == null:
    return

  visit(node, node_depth)
  for (TreeNode child in node.children):
    dfs(child, node_depth + 1)
    visit(node, node_depth)

# Save a node's depth, inverse mapping and
# position in the Euler tour
function visit(node, node_depth):
  nodes[tour_index] = node
  depth[tour_index] = node_depth
  last[node.index] = tour_index
  tour_index = tour_index + 1
```

```
# Eulerian tour index position
tour_index = 0

# Do an Eulerian Tour of all the nodes using
# a DFS traversal.
function dfs(node, node_depth = 0):
  if node == null:
    return

  visit(node, node_depth)
  for (TreeNode child in node.children):
    dfs(child, node_depth + 1)
    visit(node, node_depth)

# Save a node’s depth, inverse mapping and
# position in the Euler tour
function visit(node, node_depth):
  nodes[tour_index] = node
  depth[tour_index] = node_depth
  last[node.index] = tour_index
  tour_index = tour_index + 1
```

```
# Eulerian tour index position
tour_index = 0

# Do an Eulerian Tour of all the nodes using
# a DFS traversal.
function dfs(node, node_depth = 0):
  if node == null:
    return

  visit(node, node_depth)
  for (TreeNode child in node.children):
    dfs(child, node_depth + 1)
    visit(node, node_depth)

# Save a node's depth, inverse mapping and
# position in the Euler tour
function visit(node, node_depth):
  nodes[tour_index] = node
  depth[tour_index] = node_depth
  last[node.index] = tour_index
  tour_index = tour_index + 1
```

```
# Eulerian tour index position
tour_index = 0

# Do an Eulerian Tour of all the nodes using
# a DFS traversal.
function dfs(node, node_depth = 0):
  if node == null:
    return

  visit(node, node_depth)
  for (TreeNode child in node.children):
    dfs(child, node_depth + 1)
    visit(node, node_depth)

# Save a node’s depth, inverse mapping and
# position in the Euler tour
function visit(node, node_depth):
  nodes[tour_index] = node
  depth[tour_index] = node_depth
  last[node.index] = tour_index
  tour_index = tour_index + 1
```

```
# Eulerian tour index position
tour_index = 0

# Do an Eulerian Tour of all the nodes using
# a DFS traversal.
function dfs(node, node_depth = 0):
  if node == null:
    return

  visit(node, node_depth)
  for (TreeNode child in node.children):
    dfs(child, node_depth + 1)
    visit(node, node_depth)

# Save a node's depth, inverse mapping and
# position in the Euler tour
function visit(node, node_depth):
  nodes[tour_index] = node
  depth[tour_index] = node_depth
  last[node.index] = tour_index
  tour_index = tour_index + 1
```

```
# Eulerian tour index position
tour_index = 0

# Do an Eulerian Tour of all the nodes using
# a DFS traversal.
function dfs(node, node_depth = 0):
  if node == null:
    return

  visit(node, node_depth)
  for (TreeNode child in node.children):
    dfs(child, node_depth + 1)
    visit(node, node_depth)

# Save a node's depth, inverse mapping and
# position in the Euler tour
function visit(node, node_depth):
  nodes[tour_index] = node
  depth[tour_index] = node_depth
  last[node.index] = tour_index
  tour_index = tour_index + 1
```

```
# Query the Lowest Common Ancestor (LCA) of
# the two nodes with the indices `index1` and
# `index2`.
function lca(index1, index2):

  l = min(last[index1], last[index2])
  r = max(last[index1], last[index2])

  # Do RMQ to find the index of the minimum
  # element in the range [l, r]
  i = sparse_table.queryIndex(l, r)

  # Return the TreeNode object for the LCA
  return nodes[i]
```

```
# Query the Lowest Common Ancestor (LCA) of
# the two nodes with the indices `index1` and
# `index2`.
function lca(index1, index2):

  l = min(last[index1], last[index2])
  r = max(last[index1], last[index2])

  # Do RMQ to find the index of the minimum
  # element in the range [l, r]
  i = sparse_table.queryIndex(l, r)

  # Return the TreeNode object for the LCA
  return nodes[i]
```

```
# Query the Lowest Common Ancestor (LCA) of
# the two nodes with the indices `index1` and
# `index2`.
function lca(index1, index2):

  l = min(last[index1], last[index2])
  r = max(last[index1], last[index2])

  # Do RMQ to find the index of the minimum
  # element in the range [l, r]
  i = sparse_table.queryIndex(l, r)

  # Return the TreeNode object for the LCA
  return nodes[i]
```

```
# Query the Lowest Common Ancestor (LCA) of
# the two nodes with the indices `index1` and
# `index2`.
function lca(index1, index2):

  l = min(last[index1], last[index2])
  r = max(last[index1], last[index2])

  # Do RMQ to find the index of the minimum
  # element in the range [l, r]
  i = sparse_table.queryIndex(l, r)

  # Return the TreeNode object for the LCA
  return nodes[i]
```

```
# Query the Lowest Common Ancestor (LCA) of
# the two nodes with the indices `index1` and
# `index2`.
function lca(index1, index2):

  l = min(last[index1], last[index2])
  r = max(last[index1], last[index2])

  # Do RMQ to find the index of the minimum
  # element in the range [l, r]
  i = sparse_table.queryIndex(l, r)

  # Return the TreeNode object for the LCA
  return nodes[i]
```

Unused slides follow

Popular LCA methods on static trees

1. Find the **Eulerian Tour** of a rooted tree, and subsequently do Range Minimum Queries to find the LCA. Requires **O(nlogn)** preprocessing with a Sparse Table, and gives **O(1)** LCA queries

Popular LCA methods on static trees

1. Find the **Eulerian Tour** of a rooted tree, and subsequently do Range Minimum Queries to find the LCA. Requires **O(nlogn)** preprocessing with a Sparse Table, and gives **O(1)** LCA queries

2. **Tarjan’s offline LCA algorithm**. This algorithm uses a union find to find the LCA between two nodes, but requires all LCA queries to be specified in advance.

Popular LCA methods on static trees

1. Find the **Eulerian Tour** of a rooted tree, and subsequently do Range Minimum Queries to find the LCA. Requires **O(nlogn)** preprocessing with a Sparse Table, and gives **O(1)** LCA queries

2. **Tarjan's offline LCA algorithm.** This algorithm uses a union find to find the LCA between two nodes, but requires all LCA queries to be specified in advance.

3. Use the **Heavy-Light Decomposition** technique to break a tree into disjoint chains, and use this structure to do LCA queries.

Popular LCA methods on static trees

1. Find the **Eulerian Tour** of a rooted tree, and subsequently do Range Minimum Queries to find the LCA. Requires **O(nlogn)** preprocessing with a Sparse Table, and gives **O(1)** LCA queries

2. **Tarjan’s offline LCA algorithm.** This algorithm uses a union find to find the LCA between two nodes, but requires all LCA queries to be specified in advance.

3. Use the **Heavy-Light Decomposition** technique to break a tree into disjoint chains, and use this structure to do LCA queries.

4. **Farach-Colton and Bender technique** improves on the Euler Tour + RMQ solution by reducing the pre-processing time to **O(n)**.

Popular LCA methods on static trees

1. Find the **Eulerian Tour** of a rooted tree, and subsequently do Range Minimum Queries to find the LCA. Requires **O(nlogn)** preprocessing with a Sparse Table, and gives **O(1)** LCA queries

2. **Tarjan's offline LCA algorithm.** This algorithm uses a union find to find the LCA between two nodes, but requires all LCA queries to be specified in advance.

3. Use the **Heavy-Light Decomposition** technique to break a tree into disjoint chains, and use this structure to do LCA queries.

4. **Farach-Colton and Bender technique** improves on the Euler Tour + RMQ solution by reducing the pre-processing time to **O(n)**.

5. … and several more algorithms like **Binary Lifting** and the naive approach of walking up the tree.

Popular LCA methods on static trees

1. Find the **Eulerian Tour** of a rooted tree, and subsequently do Range Minimum Queries to find the LCA. Requires **O(nlogn)** preprocessing with a Sparse Table, and gives **O(1)** LCA queries

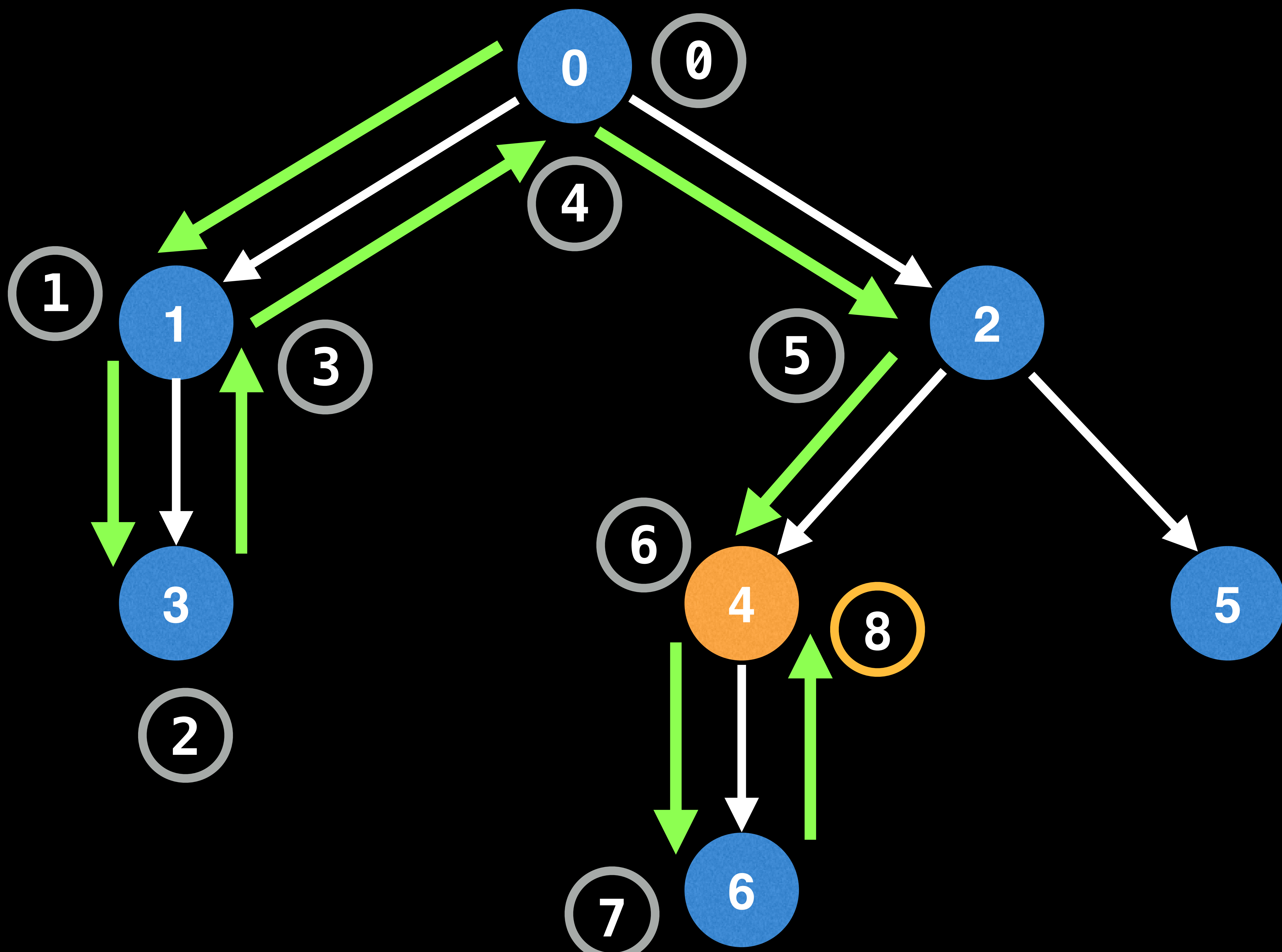
2. **Tarjan’s offline LCA algorithm.** This algorithm uses a union find to find the LCA between two nodes, but requires all LCA queries to be specified in advance.

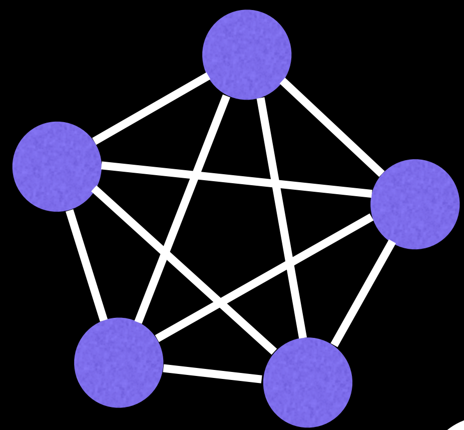
3. Use the **Heavy-Light Decomposition** technique to break a tree into disjoint chains, and use this structure to do LCA queries.

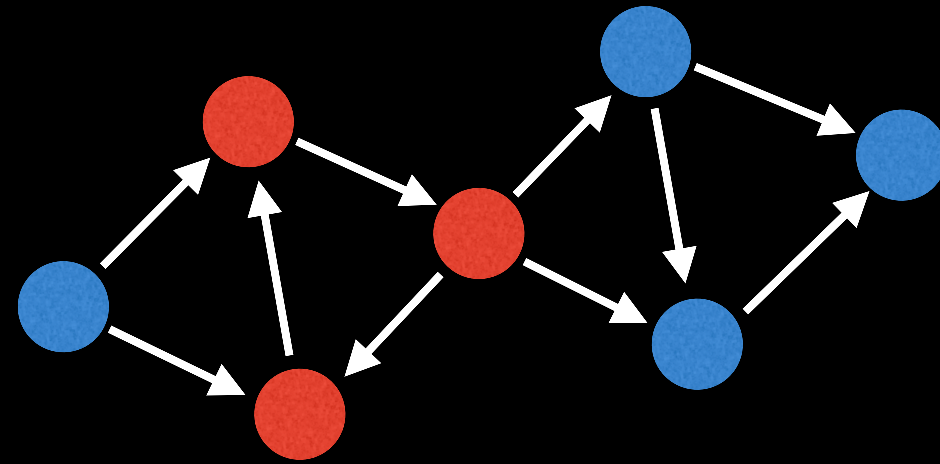
4. **Farach-Colton and Bender technique** improves on the Euler Tour + RMQ solution by reducing the pre-processing time to **O(n)**.

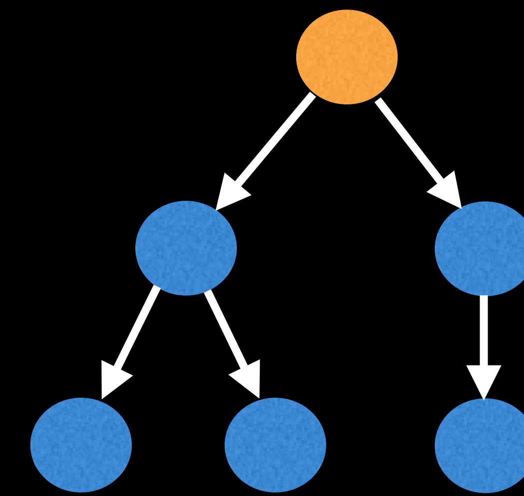
5. … and several more algorithms like **Binary Lifting** and the naive approach of walking up the tree.

Lowest Common Ancestor

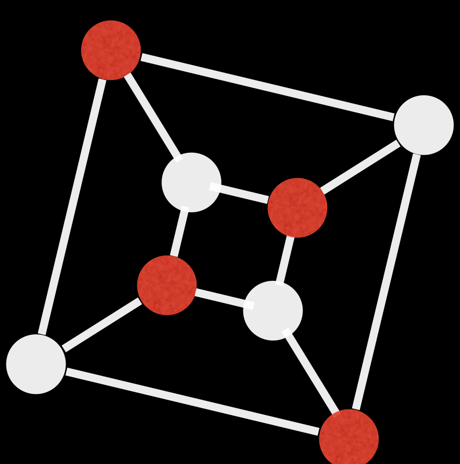


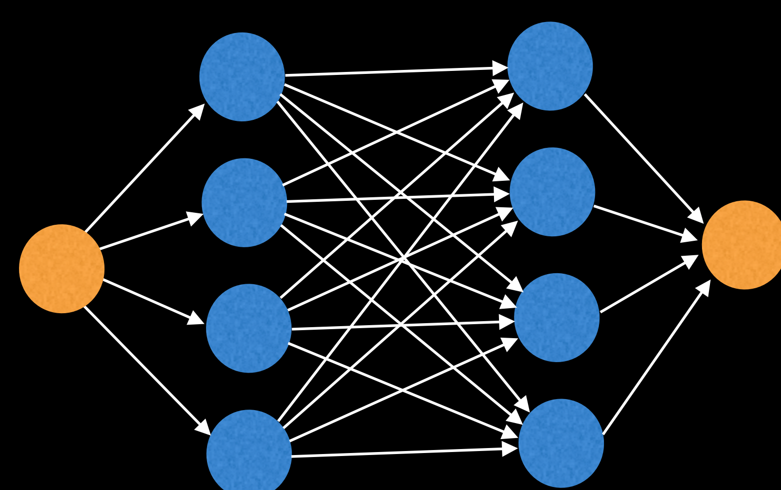


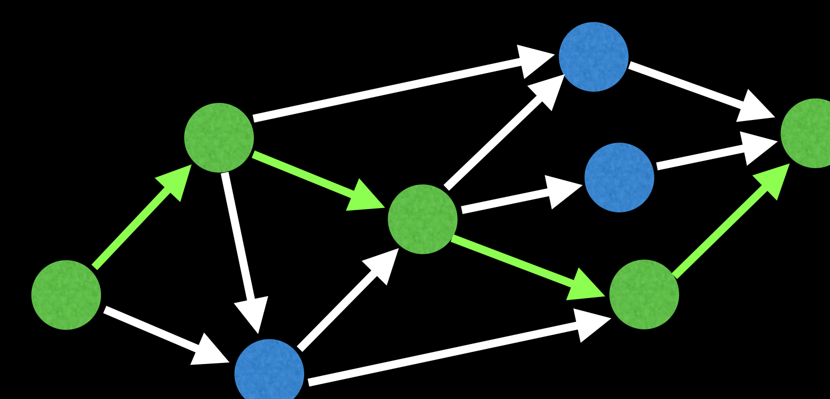




Graph Theory Video Series







Lowest Common Ancestor source code

Eulerian tour + range minimum query method

William Fiset

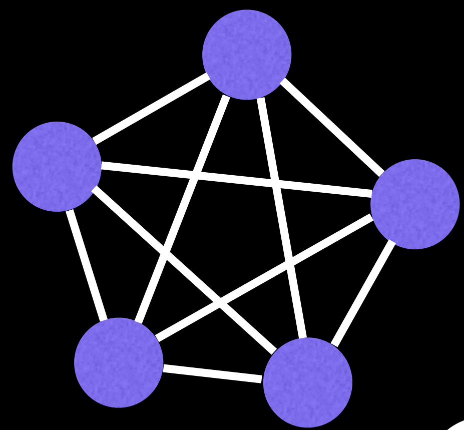
Previous video explaining the LCA problem:

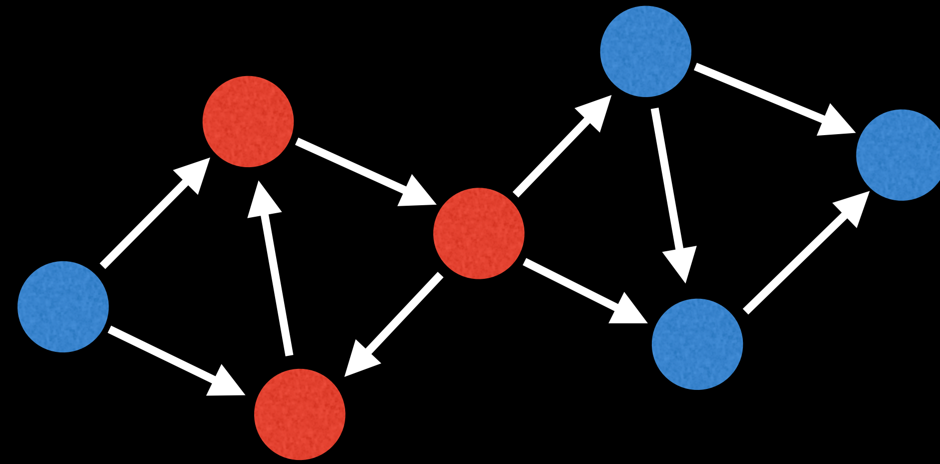
Source Code Link

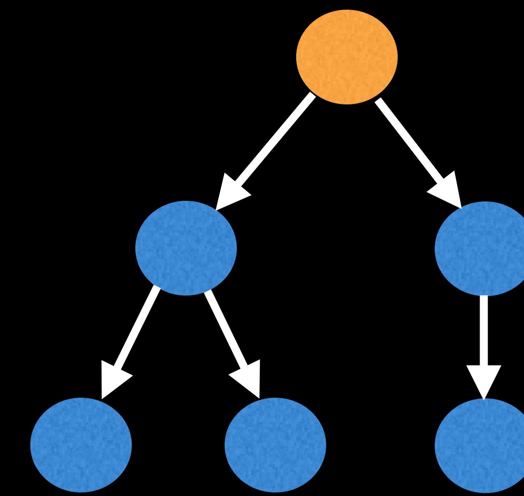
Implementation source code can be found at the following link:

github.com/williamfiset/algorithms

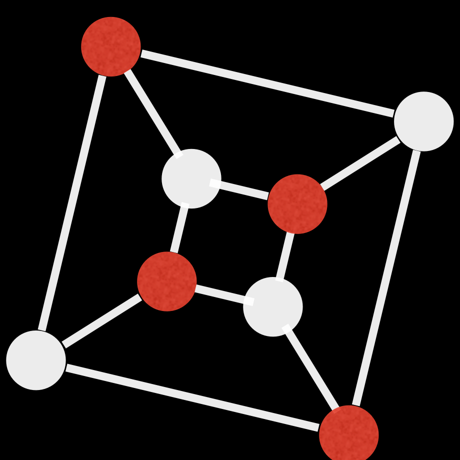
Link in the description below:

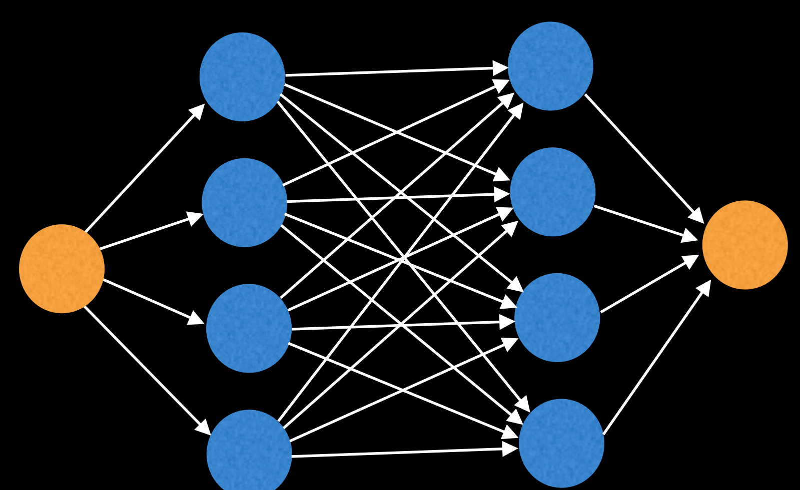


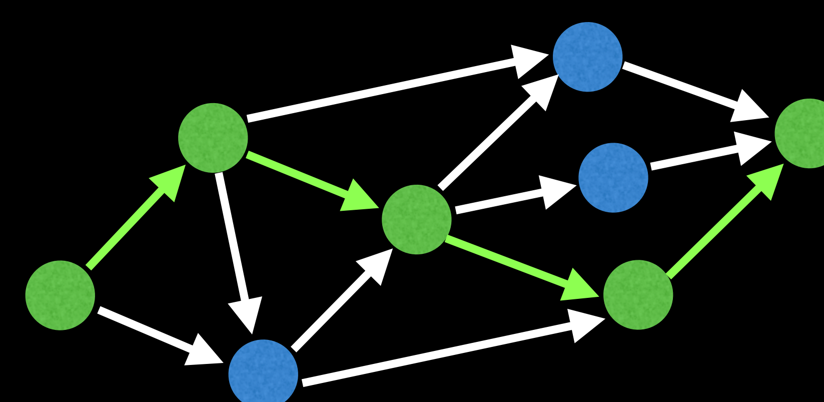




Graph Theory Video Series



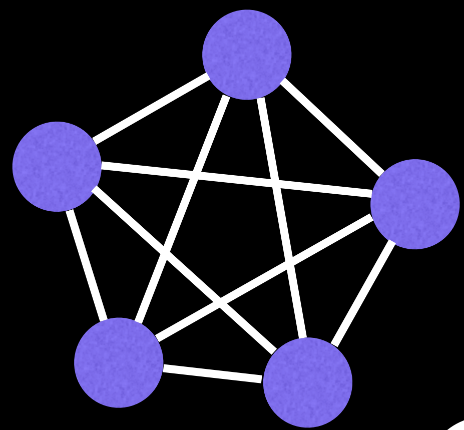


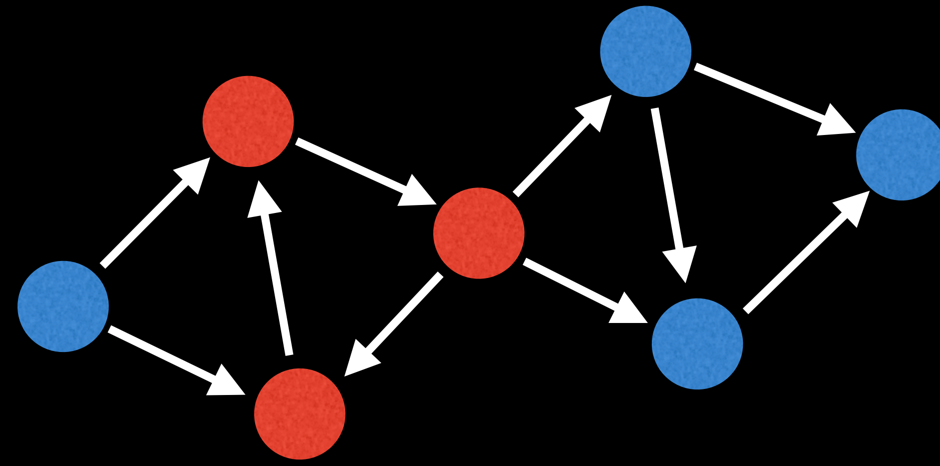


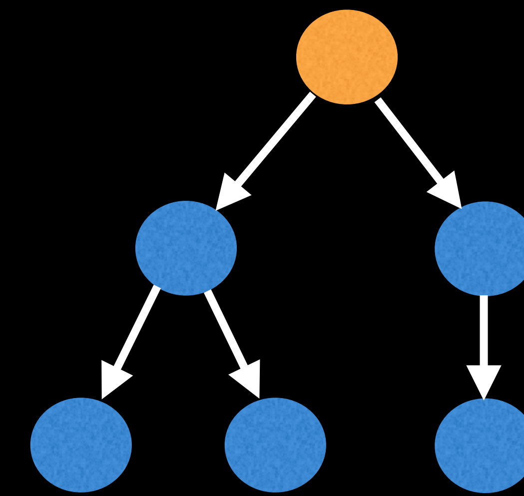
Heavy-Light Decomposition

(Heavy path decomposition)

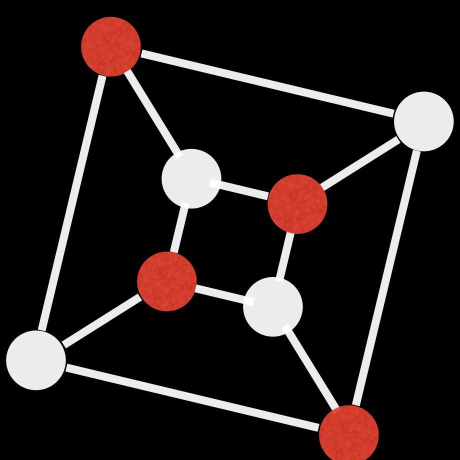
Micah Stairs | William Fiset

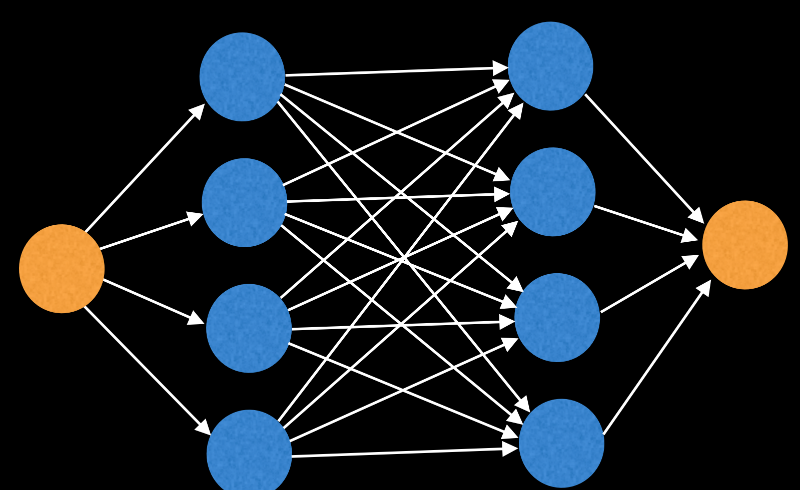


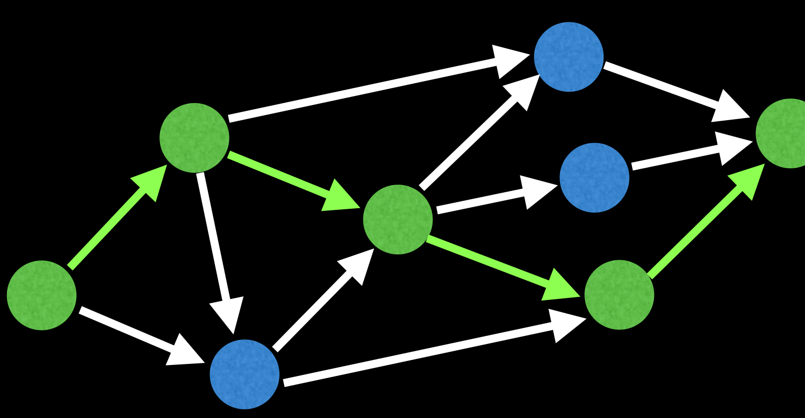




Graph Theory Video Series







Tree Centroid Decomposition

William Fiset